An Implementation of Reynolds' Defunctionalization Method for a Modern Functional Language

Jeffrey M. Bell

A thesis submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science and Engineering

January 1994

The thesis "An Implementation of Reynolds' Defunctionalization Method for a Modern Functional Language" by Jeffrey M. Bell has been examined and approved by the following Examination Committee:

James Hook
Assistant Professor
Thesis Research Advise

Tim Sheard
Assistant Professor

Françoise Bellegarde
Research Associate Professor

Acknowledgements

Thanks go to my advisor, Jim Hook, for his positive encouragement and help. I also thank my thesis committee, Françoise Bellegarde and Tim Sheard, for taking the time to make a contribution to my thesis.

Many fellow students deserve thanks as well. I'd especially like to thank Jeff Lewis for helping me with technical problems and making silly jokes, David Burke for helping me with partial evaluation, and Tim Linger for innocently pointing out a major flaw in my program when I thought I was almost done (although I didn't feel grateful at the time). Michael Gerlek, my juggling partner, also helped provide stress relief and much-needed work breaks.

Most of all I would like to thank my fiance Andrea for her encouragement, love and support, and my parents for more of the same (not to mention the money!).

Contents

List of Tables

List of Figures

Abstract

An Implementation of Reynolds' Defunctionalization Method for a Modern Functional Language

Jeffrey M. Bell, M.S.
Oregon Graduate Institute of Science & Technology, 1994

Supervising Professor: James Hook

In a 1972 paper, John C. Reynolds outlined a method for removing higher order functions from a functional language program by representing functional values as data and interpreting the representations as needed. This transformation is known as *defunctionalization.* This thesis describes an automated defunctionalization system that transforms a significant portion of core Standard ML syntax. Modern functional language features addressed by this research include pattern matching, parametric polymorphism, "multiple alternate" expressions, local declarations, and function values contained in data structures. This research elaborates on details not explained in earlier presentations of the defunctionalization method, including detection of higher order functions, function values that are "shared" by several higher order functions, and higher order functions with higher order parameters.

Chapter 1

Introduction

1.1 Higher Order Functions and Their Elimination

Higher order functions (HOFs) are either

1. functions with arguments that are themselves functions
2. functions that return function values

Fold functions are common examples of HOFs of the first sort, and curried functions are examples of the second sort.

Defunctionalization [Rey72], *firstification* [Nel91], or *higher-order removal* [CD93] is the process of eliminating higher order functions from a program, while preserving the semantics of the program. It is desirable to eliminate higher order functions from a program because some program transformations which improve program efficiency can only be performed on first order programs (*i.e.* programs without higher order functions).

Higher order functions of the second sort can be eliminated from a program via uncurrying. The uncurrying transformation ensures that every function call includes values for all arguments to the function, and that function declarations include all necessary parameters. If a value for one of a function's arguments is not available at a call site, a lambda abstraction (*section*) is constructed at the call site, supplying the function with all its arguments.

This research focuses on HOFs of the first sort, *i.e.* functions with function valued parameters (FVPs). It is assumed that curried functions are eliminated via uncurrying

before the program is transformed by the defunctionalization system described in this thesis. In fact, a "stronger" form of uncurrying is assumed to have been done to input programs for the defunctionalization system. This transformation ensures that each function has only one argument, which may be a tuple.

1.2 HOF Elimination via Specialization

Chin and Darlington [CD93] have developed a HOF elimination algorithm based on fold/unfold rules that uses *specialization*. The following example demonstrates the algorithm. The program below contains a HOF called *map*, which applies a function to each argument in a list. After its definition, *map* is called with the function *increment*.

$$\begin{aligned}&\textbf{fun}\ \ map\ f\ Nil\ =\ Nil\\ &\quad \mid\ map\ f\ Cons(x,xs)\ =\ Cons(f\ x, map\ f\ xs)\end{aligned}$$

$$\textbf{fun}\ addone\ l\ \ =\ map\ increment\ l$$

After specialization, the call to *map* is unfolded to the body of *map*, specialized with respect to the FVP used. The resulting program is:

$$\begin{aligned}&\textbf{fun}\ \ map\ f\ Nil\ =\ Nil\\ &\quad \mid\ map\ f\ Cons(x,xs)\ =\ Cons(f\ x, map\ f\ xs)\end{aligned}$$

$$\begin{aligned}&\textbf{fun}\ \ addone\ Nil\ \ =\ Nil\\ &\quad \mid\ addone\ Cons(x,xs)\ =\ Cons(increment\ x, addone\ xs)\end{aligned}$$

Note that *map* is not modified. However, since it is no longer called, it can safely be eliminated from the program.

This method is limited. Only higher order functions whose functional arguments are *variable-only* can be specialized with this approach. A functional argument in a recursive call to a higher order function is variable-only if the argument is an identifier. Functional

arguments in non-recursive function calls are all variable-only. If a functional argument in a recursive call to a higher order function is not variable-only, the algorithm will go into an infinite loop trying to specialize the function.

1.3 Reynolds' Defunctionalization Method

In a 1972 paper, John C. Reynolds outlined a more general method for eliminating higher order functions known as defunctionalization [Rey72]. The key to this method is representing FVPs as data values. When FVPs are transformed into data values, higher order functions become first order, since they no longer have functions as arguments. However, applications of the FVPs need to be modified, since a data value cannot be applied as a function. This is accomplished by creating an "apply" function for each FVP of a HOF. The apply function is called wherever the FVP was applied in the original HOF. The apply function takes as arguments the data value representing the FVP and all the arguments to the FVP. The apply function decodes the FVP representation, and applies the appropriate function to the remaining arguments.

The following example illustrates Reynolds' defunctionalization method. Consider the following program, which contains a higher order function *high* and two calls to *high*.

$$\textbf{fun}\ high\ (somefun, x)\ =\ somefun\ x$$

$$\textbf{val}\ two\ =\ high\ (increment, 1)$$

$$\textbf{val}\ zero\ =\ high\ (decrement, 1)$$

If the program is defunctionalized using strings containing the function name as the representation of function values, the following program is the result.

```
fun apply_somefun (function_rep, function_arg) =
    case function_arg of
        "increment" ⇒ increment function_arg
      | "decrement" ⇒ decrement function_arg

fun high (somefun, x) = apply_somefun (somefun, x)

val two = high ("increment", 1)

val zero = high ("decrement", 1)
```

Reynolds' method has been used in many papers to defunctionalize denotational semantics specifications. Motivations for this include detection of stack-based environments [Sch88], detection of global variables [Sch85a], and deriving an implementation of the specification [BJ82, Sch85b, Wan85].

Reynolds' method is more powerful than the specialization method of Chin and Darlington because it is not limited to variable-only function valued arguments. However, a defunctionalized program is only *pseudo first-order* [CD93] because it still mimics the runtime characteristics of the original higher order program. Chin and Darlington's method produces more efficient implementations. Therefore, when either method is applicable, it is desirable to use specialization rather than defunctionalization.

1.4 Limitations of Reynolds' Method

Reynolds presents this method in an *ad hoc* way, via a specific example. This is also true in examples of the method in other papers. Thus there are many details left out of these presentations. These details are needed for a fully automated implementation of the defunctionalization method. In particular, the following details are not explicitly discussed in the paper:

D-1 A method for detecting HOFs and FVPs in a program

D-2 Locally declared higher order functions (via **let** or lambda expressions)

D-3 Function values declared in a scope more local than the HOF to which the function value is a FVP

D-4 FVPs that are shared by more than one HOF (e.g., when one HOF passes its FVP to another HOF)

D-5 HOFs with HOFs as parameters

Likewise, many features of modern functional languages are not explicitly dealt with in Reynolds' method. These include:

F-1 Pattern matching

F-2 Polymorphic higher order functions

F-3 Function values contained within data structures

F-4 Complications to the method arising from use of **case** and similar expressions

1.5 Firstify—An Automated Defunctionalization System

The research presented in this thesis is an attempt to produce an automated defunctionalization system for a modern functional language. This required the identification and resolution of the above limitations. *Firstify*, the result of this effort, is a fully automated implementation of Reynolds' method.

Firstify addresses all of the limitations outlined above, at least to some extent. In particular, D-1, D-4, F-1, F-2[1] , and F-4 are fully addressed. Issues D-5 and F-3 are partially addressed. Solutions to D-2 and D-3 have been developed, but are only partially addressed in the implementation. Since another relatively simple transformation can eliminate **let** expressions and nearly all lambda abstractions from a program, it was not necessary to include full support for these features. However, they are interesting

[1] Addressing this feature required the use of type information, and is in fact the only use for type information in *Firstify*. It is therefore believed that this research applies to untyped languages as well.

subproblems. Other minor restrictions to *Firstify* input are listed in the final chapter of this thesis.

The input and output to *Firstify* are in the form of Standard ML programs, expressed as lists of declarations in the CRML abstract syntax. CRML, or Compile-Time Reflective ML [She93], is a reflective dialect of ML which allows ML declarations, expressions, patterns, and types to be represented as data. *Firstify* is written in CRML.

1.6 Problem Examples

This section outlines problems addressed in this thesis by showing an example illustrating each problem. A brief description of each problem is included, along with the chapter or section number in which the problem is addressed. Solutions to the problems are described in the referenced chapters and sections.

1.6.1 Identifier Aliasing

By binding identifiers to other identifiers, the name of HOF or function passed as a FVP might be aliased as another identifier. This poses problems when looking for applications of FVPs and transforming calls to HOFs, because the "real" name of the function is needed. This problem is addressed in Section 2.1. Example:

```
fun high (f,x) =
    let val g = f
    in g x end

val two = case increment of myinc => high (myinc, 1)
```

1.6.2 HOFs With Shared FVPs

Higher order functions *share* FVPs if both HOFs are passed the same FVP. One way for HOFs to share FVPs is if one HOF passes a FVP to another HOF in a function call. All HOFs that share FVPs must also share an apply function. There are four ways in which HOFs with shared FVPs are detected. This problem is addressed in Section 3.4.

Example 1, in which the first parameter of *high* and the second parameter of *high'* are shared FVPs. The shared FVPs are detected when transforming the body of the HOF *high*.

$$\textbf{fun}\ high\ (f,x)\ =\ (high'\ (x,f),\ f\ x)$$

Example 2, in which the second and third parameters of *high* are shared FVPs. The shared FVPs are detected when transforming the body of the HOF *high*

$$\begin{aligned}\textbf{fun}\ high\ (f,g,x)\ =\ (&\textbf{case}\ x\ \textbf{of}\\ &\quad nil\ \Rightarrow\ f\\ &\mid\ x :: xs\ \Rightarrow\ g)\ x\end{aligned}$$

Example 3, in which *outer* and *high* share their first (function-valued) parameter. The shared FVPs are detected when transforming calls to *high*.

$$\textbf{fun}\ outer\ (f,x)\ =\ high\ (f,x)$$

Example 4, in which *high* and *high'* share their first (function-valued) parameter. The shared FVPs are detected when transforming calls to *high*.

$$\begin{aligned}\textbf{fun}\ call\_high\ x\ =\ (&\textbf{case}\ x\ \textbf{of}\\ &\quad nil\ \Rightarrow\ high\\ &\mid\ x :: xs\ \Rightarrow\ high')\ (f,x)\end{aligned}$$

1.6.3 Lambda Abstractions as FVPs

Lambda abstractions can be passed to HOFs. If the lambda abstraction uses variables bound locally, the representation of the FVP as a data value must encode the values of locally bound variables. This problem is addressed in Section 3.6. Example:

$$\textbf{fun}\ call\_high\ x\ =\ high\ (\lambda z.g(x,y,z),0)$$

1.6.4 Multiple Alternate Expressions

Some expressions, like **case**, contain several explicit possible values. When one of these *multiple alternate expressions* is an expression that needs to be transformed by *Firstify*, the transformation becomes more complex. This problem is addressed in Section 3.7.

Example 1, a multiple alternate expression in the body of a HOF:

```
fun high (f,x) = (case x of
                    nil ⇒ id
                  | x :: xs ⇒ f) x
```

Example 2, a multiple alternate expression used as the HOF expression at a HOF call site:

```
fun call_high x = (case x of
                     nil ⇒ high
                   | x :: xs ⇒ high') (f,x)
```

Example 3, a multiple alternate expression used as the FVP expression at a HOF call site:

```
fun call_high x = high (case x of
                          0 ⇒ increment
                        | _ ⇒ decrement,x)
```

1.6.5 Local Declarations

When HOFs or functions used as FVPs are declared at a local scope, the transformations must utilize scope information. This problem is addressed in Section 4.2. Example:

```
fun outer x =
    let fun high (f,x) = f x
        fun increment x = x + 1
    in
        high (increment,x)
    end
```

1.6.6 FVP and HOF Declared at Different Scopes

Suppose a FVP passed to a HOF is declared at a scope that is more "inner" than the scope at which the HOF is declared. Since the apply function for the HOF must be

declared after the FVP is declared but before the HOF is declared, the apply function cannot be correctly inserted into the program. This problem is addressed in Section 4.3.1. Example:

```
fun high (f, x) = f x

fun comma_separate (x, y) =
    let fun concat_to_x s = x ^ "," ^ s
    in high (concat_to_x, y) end
```

1.6.7 Polymorphic Higher Order Functions

If a HOF is polymorphic in the type of its FVP and the HOF is instantiated at more than one type, then the apply function for the HOF will not be well-typed. Instead, an apply function is needed for each type at which the HOF is instantiated. This problem is addressed in Chapter 5. Example:

```
fun plus1 (x : int) = x + 1
fun java s          = s ^ "java"
fun high (f, x)     = f x
val two             = high (plus1, 1)
val coffee          = high (java, "mocca-")
```

1.6.8 Functions in Data Structures

Function values contained in data structures (*i.e.* records, tuples, and datatype values) pose unique problems in defunctionalization. This problem is discussed in Section 6.1. Example:

```
datatype Changes =
    ChangeWith of int → int
  | Unchanged

fun change_number (change, x) =
    case change of
        ChangeWith f ⇒ f x
      | Unchanged ⇒ x

val two = change_number (ChangeWith increment, 1)
```

1.6.9 HOFs with Higher Order FVPs

A higher order function may have a function valued parameter that is itself a higher order function. In this case, the HOF passed as a FVP may never be explicitly called in the program, but only called in the body of the other HOF. This problem is discussed in Section 6.2. Example:

```
fun high (f, x) = f x
fun higher (f, g, x) = f (g, x)
val call_higher = higher(high, id, 1)
```

1.6.10 Transforming FVP Values Away from the Call Site

In some situations, the transformation of a function value into a data value needs to take place at some place in the program other than at the HOF call site. This problem is discussed in Section 6.3. Example:

```
val params = (increment, 1)
    ⋮
val call_high = high params
```

Chapter 2

Non-Operational Implementation Overview

The functionality of *Firstify* can be described in terms of three major subproblems. This is not an operational breakdown, since the subproblems do not correspond to phases of execution, but it is useful in that it clarifies the functionality of *Firstify*.

2.1 Subproblem I—Maintaining Information

To perform transformations on a program, and even to determine when and if a transformation is necessary, some information about identifiers in the program is needed. In particular, *Firstify* needs information about identifiers referring to function values that might be passed as a FVP and identifiers referring to parameters to see if a parameter is applied as a function, and the surrounding function is thus higher order.

Firstify maintains this information in an *environment*, that gives bindings for identifiers. Environments are realized as functions from identifiers (strings) to environment values in the *enventry* datatype, shown in Figure 2.1. Environments are dynamic, changing as the structure of a program is traversed. For example, when a function declaration is encountered, the function name is entered in the environment, bound to the information that the identifier is a function name. Also, all identifiers in the formal parameter[1] to the function are bound to information indicating the position of the identifier in the formal

[1]There is only one formal parameter because functions are assumed to be "strongly" uncurried.

```
datatype enventry =
    NotInEnv
  | NotApplicable
  | RefTo of string
  | Param of function_id * parameter_id
  | LocalDec of function_id
  | TuplePartOf of int * enventry
  | ConstrPartOf of enventry
  | RecordPartOf of enventry
  | BasedOn of enventry
  | TupleContaining of enventry list
  | RecordContaining of (string * enventry)list
  | PossibleValues of enventry list
  | Closure of environ * int * ((string pattern * (environ -> enventry)) list)
```

Figure 2.1: The *enventry* datatype

parameter. The formal parameter bindings are in the environment used while traversing the function body, but not in the environment used while traversing the remaining declarations in the program. However, the binding for the function name remains in the environment used while traversing all other declarations at the same scope.

In addition to environment information, *Firstify* needs to maintain type information about functions in the program. This information is necessary to perform specialization on HOFs (see Chapter 5). Type information is obtained at the beginning of the transformation process, as described in Section 5.3. Type information is updated as specialization occurs and the types of HOFs change due to transformations. While type information could have been included in environments (in the *LocalDec* constructor), it is more efficient and convenient to gather all type information at one time.

A description of how each *enventry* constructor is used to represent environment bindings is shown in Appendix A. Tables 2.1 and 2.2 show an illustrative subset of this information. Note that arguments to the *TuplePartOf* constructor encode information

about the position of an identifier in a tuple. However, similar information is not contained in *ConstrPartOf* and *RecordPartOf* arguments. This is because it is assumed that in the input to *Firstify*, function values are not contained in record or datatype values, and environment bindings are primarily used to obtain information about function values. This is discussed further in Section 6.1.

Building and maintaining environments is primarily done via functions that calculate environment values for values in the CRML datatypes that represent expressions and declarations. A description of these datatypes and a more complete explanation of the environment-calculating functions is given in Chapter 3. The environment-calculating function for declarations returns a list of identifiers declared, each of which is paired with the *enventry* value for the identifier. The environment-calculating function for expressions returns the *enventry* value to which the expression corresponds.

When a pattern is matched to an expression in the source program, the identifiers in the pattern need to be bound to the *enventry* value of the expression. This is trivial when the pattern consists of a single identifier. When the pattern is more complex, the *bind* function computes a list of bindings for identifiers in a pattern to an *enventry* value. Figure 2.2 shows the definition of *bind*. The notation $\langle\langle x \rangle\rangle$ indicates the representation of the program fragment x, as opposed to the value of x.

2.2 Subproblem II—Determining Transformations to Perform

Higher order functions are found by looking for function parameters that are applied as functions. However, when a HOF and FVP are found before the transformations can be performed, all other HOFs with FVPs that are *shared* with the detected FVP must be found, so that the transformations for all the HOFs can be performed at once. One way for two FVPs to be shared is if a HOF passes its FVP to another HOF. If two HOFs with shared FVPs were transformed independently of one another, and the representation chosen for the FVPs of the two functions were not the same type, then the program

Constructor	Domain Description of Use Example
RefTo	*string* Identifier directly referring to another identifier **let val** x $=$ y **in** *BODY* **end** While processing *BODY*, x is bound to *RefTo "y"*.
Param	*function_id* $*$ *parameter_id* Part of a parameter to a function. The *function_id* encodes which function, and the *parameter_id* encodes the position of the identifier in the function's parameter. **fun** f x $=$ *BODY* While processing *BODY*, x is bound to *Param* ((*"f"*, 0), *Position* 0). The types *function_id* and *parameter_id* are discussed in Section 3.3.1.
LocalDec	*function_id* Identifier declared in a **val**, **fun**, **datatype**, **exception**, or **type** declaration. The *function_id* encodes information about the identifier, including the scope at which it is declared. **fun** f ... While processing all declarations occurring after the declaration of f, f is bound to *LocalDec* (*"f"*, 0). The *function_id* type is discussed in Section 3.3.1.

Table 2.1: Uses of *enventry* constructors (part 1)

Constructor	Domain Description of Use Example
TuplePartOf	*int * enventry* Identifier that is contained in a tuple. The *enventry* represents the tuple in which the identifier is contained, and the *int* encodes the position of the identifier in the tuple. **val** $z = a$ **let val** $(x, y) = z$ **in** *BODY* **end** While processing *BODY*, x is bound to *TuplePartOf* $(1, RefTo\ \text{“}a\text{”})$ and y is bound to *TuplePartOf* $(2, RefTo\ \text{“}a\text{”})$.
TupleContaining	*enventry list* Identifier is a tuple, containing several values. The values within the tuple are represented in order in the *enventry list.* **val** $y = a$ **val** $z = b$ **let val** $x = (y, z)$ **in** *BODY* **end** While processing *BODY*, x is bound to *TupleContaining* $[RefTo\ \text{“}a\text{”}, RefTo\ \text{“}b\text{”}]$.
PossibleValues	*enventry list* Identifier refers to several possible values, such as when a **case** or **handle** expression is used. Each *enventry* in the list represents one possible binding for the identifier. **val** $y = a$ **val** $z = b$ **let val** $x =$ **case** c *of* $0 \Rightarrow y$ $\mid \_ \Rightarrow z$ **in** *BODY* **end** While processing *BODY*, x is bound to *PossibleValues* $[RefTo\ \text{“}a\text{”}, RefTo\ \text{“}b\text{”}]$.

Table 2.2: Uses of *enventry* constructors (part 2)

$$
\begin{array}{lcl}
bind\ \langle\langle x \rangle\rangle & e = & [\text{“}x\text{”} \xrightarrow{b} e] \\
bind\ \langle\langle c \rangle\rangle & e = & nil \\
bind\ \langle\langle (C\ x) \rangle\rangle & e = & bind\_wrap\ \langle\langle x \rangle\rangle\ e\ ConstrPartOf \\
bind\ \langle\langle (x, y) \rangle\rangle & e = & \textbf{case}\ e\ \textbf{of} \\
 & & \quad TupleContaining\ [e_1, e_2]\ \Rightarrow \\
 & & \quad\quad (bind\ \langle\langle x \rangle\rangle\ e_1)\ @\ (bind\ \langle\langle y \rangle\rangle\ e_2) \\
 & & \quad |\ e'\ \Rightarrow \\
 & & \quad\quad (bind\_wrap\ \langle\langle x \rangle\rangle\ e'\ (\lambda e.\, TuplePartOf\ (1, e)))\ @ \\
 & & \quad\quad (bind\_wrap\ \langle\langle y \rangle\rangle\ e'\ (\lambda e.\, TuplePartOf\ (2, e))) \\
bind\ \langle\langle \{t_1 = x, t_2 = y\} \rangle\rangle & e = & \textbf{case}\ e\ \textbf{of} \\
 & & \quad RecordContaining\ [(t'_1, e_1), (t'_2, e_2)]\ \Rightarrow \\
 & & \quad\quad (bind\ \langle\langle x \rangle\rangle\ e_1)\ @\ (bind\ \langle\langle y \rangle\rangle\ e_2) \\
 & & \quad\quad\quad if\ \forall\ i,\ t_i = t'_i \\
 & & \quad |\ e'\ \Rightarrow \\
 & & \quad\quad (bind\_wrap\ \langle\langle x \rangle\rangle\ e'\ RecordPartOf)\ @ \\
 & & \quad\quad (bind\_wrap\ \langle\langle y \rangle\rangle\ e'\ RecordPartOf) \\
bind\ \langle\langle (a\ \textbf{as}\ p) \rangle\rangle & e = & (bind\ \langle\langle a \rangle\rangle\ e)\ @\ (bind\ \langle\langle p \rangle\rangle\ (RefTo\ \text{“}a\text{”})) \\
bind\ \langle\langle (x : t) \rangle\rangle & e = & bind\ \langle\langle x \rangle\rangle\ e \\
bind\ \langle\langle \_ \rangle\rangle & e = & nil \\
 & & \\
bind\_wrap\ p\ e\ wrapper & = & map^{list}\ (\lambda(a \xrightarrow{b} e).(a \xrightarrow{b} (wrapper\ e)))\ (bind\ p\ e)
\end{array}
$$

Note: $a \xrightarrow{b} e$ means identifier a is bound to environment value e. In the implementation, the binding $a \xrightarrow{b} e$ is the pair (“a”,e).

Figure 2.2: The *bind* function.

would not be well typed. Thus it is necessary to find all HOFs with shared FVPs before performing transformations on the first HOF. A collection of HOFs with shared FVPs is known as a HOF *set*. In a HOF set, the first HOF that was detected is known as the *base* HOF, and the FVP of that HOF is the *base* FVP. HOFs with shared FVPs are discussed in Section 3.4.

Another approach to detecting HOFs is type analysis. By examining the type of each function, looking for functional types in the arguments to the function, it is possible to determine if the function is higher order. This is perhaps a more efficient method for detecting higher order functions. However, the method described above works well in the control structure developed for performing transformations, described in Section 3.2.

2.3 Subproblem III—Performing the Transformations

Once a HOF set is found, the following transformations need to be performed:

1. A FVP *datatype* must be constructed. Each constructor in a FVP datatype represents a function that is passed as a FVP to a HOF in the HOF set. This datatype is inserted in the program before the declaration of the base HOF.

2. An *apply function* must be constructed that interprets values in the datatype and applies the appropriate function to its arguments. This declaration is inserted in the program before the declaration of the base HOF, but after the FVP datatype declaration.

3. At each call to one of the HOFs in the HOF set, the FVP expression must be transformed into a value representing that function. At the call site, the expression that evaluates to the HOF and the expression being passed as the FVP can be any arbitrary expression. If they are both identifiers, then the transformation is straightforward. However, the HOF expression can be a *multiple alternate expression*. A multiple alternate expression is an expression that contains several

explicit alternative values, or *alternates*. For example, a **case** statement is a multiple alternate expression. Any one of the alternates might be selected at runtime, so each alternate might need to be transformed. Likewise, the FVP expression can be a lambda abstraction or multiple alternate expression. In these instances, the transformation is more complex. Transforming multiple alternate expressions is discussed in Section 3.7 and elsewhere. Representing lambda abstractions is discussed in Section 3.6.

4. Each time a FVP is applied in the body of a HOF, the call must be transformed into a call to the apply function, passing it the FVP function representation and the arguments to the FVP function.

5. In the body of a HOF, other uses of the FVP might require transformation, since the type of the FVP has changed. For example, transformation is necessary when one alternate in a multiple alternate expression evaluates to the FVP. Since all values in all alternates of multiple alternate expressions must have unifiable types, other alternates may need to be transformed. This transformation is discussed in Section 3.7.

Chapter 3

Operational Implementation Overview

This chapter gives an operational overview of the *Firstify* implementation. It discusses the implementation language, the control structures used, the phases of execution, and other implementation issues.

3.1 CRML

CRML, or *Compile-Time Reflective ML*, is both the implementation language for *Firstify* (the *meta language*) and the language over which *Firstify* performs transformations (the *object language*). CRML is a dialect of Standard ML that includes built-in datatypes that represent program fragments in the form of CRML expressions, declarations, patterns, and types. It also includes a mechanism for compile-time reflection, that is, converting sentences in the object language into sentences in the meta language at compile time. The version of CRML used for this research includes only the core Standard ML syntax, that is, Standard ML without the module system.

There are seven datatypes used to represent program fragments. Each datatype is parameterized with a type variable α, used for the representation of identifier names. *Firstify* is concerned only with the datatypes instantiated at type *string*, since these are the datatypes used by CRML to represent program fragments. The datatypes are *erep* (expressions), *drep* (declarations), *trep* (types), *pattern* (patterns), *datadec* (individual **datatype** declarations), *fundec* (individual function declarations), and *valdec* (individual value declarations). Generally we will abbreviate type names, leaving out

the *string* parameter. The transformations discussed herein primarily use the *erep* and *drep* datatypes, explained in Tables 3.1 and 3.2.

Rather than explicitly using the datatype constructors to represent program fragments, CRML provides syntactic sugar in the form of *object brackets*. A program fragment within the object brackets $\langle\langle$ and $\rangle\rangle$ is not evaluated in the meta language, but is a value in one of the CRML datatypes (*i.e.* the object language). CRML actually provides four sets of object brackets (for expressions, declarations, patterns, and types), but we will just use $\langle\langle$ and $\rangle\rangle$ since the variety of program fragment will be apparent from the context.

Expressions within object brackets can be "escaped" with the backquote character (‘). Backquoted expressions are evaluated at compile time as meta language expressions, and the meta language value of the expression is used rather than the representation of the value. For example, $\langle\langle x + y\rangle\rangle$ represents the program fragment consisting of the expression $x + y$, but $\langle\langle\text{‘}x + y\rangle\rangle$ is some program fragment $\langle\langle E + y\rangle\rangle$, where E is the value of the variable x in the meta language. For $\langle\langle\text{‘}x + y\rangle\rangle$ to be well typed, x must be an *erep*. So if x evaluates to $App(Id\ \text{“}f\text{”}, Id\ \text{“}z\text{”})$, the program fragment represents the expression $(f\ z) + y$.

3.2 Control Structure

A *fold* function (also known as a *reduce* function or *catamorphism* [MFP91]) is a useful control structure for performing computations on values in a sum-of-products datatype. There is a fold function specific to each such datatype. The arguments to a fold function include a set of *accumulator functions* and a value in the datatype. Each accumulator function calculates over values of one constructor in the datatype. When called with a set of a accumulator functions and a value, the fold function traverses the structure of the value, performing computations (as defined by the accumulator functions) on the constituent subvalues in the value. Fold functions can be used to implement *map* functions, that compute on subvalues in a value but leave the structure of the value

Constructor	Domain Description of Use
Id	α
	Identifier. *Id "x"* represents the expression x.
Iconst	*int*
	Integer constant. *Iconst* 42 represents the expression 42.
Bconst	*bool*
	Boolean constant. *Bconst true* represents the expression *true*.
Sconst	*string*
	String constant. *Sconst "foo"* represents the expression *"foo"*.
App	α *erep* $*$ α *erep*
	Function application. *App(Id "f", Id "x")* represents the expression $f\ x$.
Tuple	α *erep list*
	Tuple expression. *Tuple* [*Id"a", Bconst false*] represents the expression $(a, false)$.
Seq	α *erep list*
	Sequence of expressions. *Seq* [*App(Id "print", Sconst "test"), Id "some_value"*] represents the expression (*print "test"; some_value*).
Record	(*string* $*$ α *erep*) *list*
	Record value. *Record* [(*"name", Sconst "Smiley"*), (*"job", Sconst "spy"*)] represents the expression $\{name = "Smiley", job = "spy"\}$.
Project	*string*
	Projection (to apply to a record value). *App(Project "name", Id "myrec")* represents the expression #*name myrec*.
Abs	(α *pattern* $*$ α *erep*) *list*
	Lambda abstraction. *Abs* [($\langle\langle nil \rangle\rangle$, *Bconst false*), ($\langle\langle x :: xs \rangle\rangle$, *Bconst true*)] represents the expression **fn** *nil* $\Rightarrow$ *false* $\mid$ $x :: xs$ $\Rightarrow$ *true*.
Letrec	α *drep* $*$ α *erep*
	let expression. *Letrec*($\langle\langle$**val** z = 17$\rangle\rangle$, *Id "z"*) represents the expression **let val** z = 17 **in** z **end**.
Case	α *erep* $*$ (α *pattern* $*$ α *erep*) *list*
	case expression. *Case(Id "l"*, [($\langle\langle nil \rangle\rangle$, *Bconst false*), ($\langle\langle x :: xs \rangle\rangle$, *Bconst true*)] represents the expression **case** *l* **of** *nil* $\Rightarrow$ *false* $\mid$ $x :: xs$ $\Rightarrow$ *true*.
Handle	α *erep* $*$ (α *pattern* $*$ α *erep*) *list*
	handle expression. *Handle*(*App(Id "hd", Id "ilist"*), [($\langle\langle Hd \rangle\rangle$, *Iconst* 0)]) represents the expression *hd ilist* **handle** *Hd* $\Rightarrow$ 0.
Raise	α *erep*
	raise expression. *Raise*(*Id "Hd"*) represents the expression **raise** *Hd*.
ExpTypeC	α *erep* $*$ α *trep*
	Explicitly typed expression. *ExpTypeC*(*Id "x"*, $\langle\langle int \rangle\rangle$) represents the expression $(x : int)$.

Table 3.1: The α *erep* Datatype, Representing Expressions

Constructor	Domain
	Description of Use
Exception	$\alpha * \alpha$ *trep*
	Exception declaration.
TypeSynonym	α *list* $* \alpha * \alpha$ *trep*
	Type synonym.
MutualVal	α *valdec list*
	Set of mutually recursive **val** declarations.
MutualFun	α *fundec list*
	Set of mutually recursive function declarations.
MutualData	α *datadec list*
	Set of mutually recursive datatype declarations.

Table 3.2: The α *drep* Datatype, Representing Declarations

intact. Fold functions can also be used to implement functions that return values of an entirely different type than the input value.

Since most of the transformations done by *Firstify* (with the exception of those that add or copy declarations) are done at the granularity of expressions, a natural control structure for performing these transformations is a fold function over *ereps*. For any of these transformations, most of the accumulator functions used by a fold would simply reconstruct the expression unmodified, with perhaps one or two accumulator functions that actually modify an expression. It would be useful to have a mechanism for maintaining a set of these "identity" accumulator functions that could be modified as needed to perform specific transformations.

Fortunately, reflection gives us such a mechanism. In the *Firstify* implementation, a set of accumulator functions is stored as a list of *ereps*. At compile time, the accumulator function expressions are evaluated and used in a fold function. To perform different transformations, one or more of the expressions in the list are replaced by new expressions representing the appropriate accumulator function.

3.2.1 Dependent Folds

Environment information is needed to perform all transformations. This information can be computed by a fold. It is desirable to combine the environment-computing fold with an expression-transforming fold in such a way that environment information is available to the transformation computation. Such a combination is possible with a *dependent fold*, or *depfold*.

A depfold takes two sets of accumulator functions and a value and essentially computes two folds in parallel on the value. Thus, a dependent fold returns a pair of values, each of which is the result of a fold. In a normal fold function, the accumulator functions compute with values of the same type as the result type of the fold, wherever a recursive reference to the datatype occurs. However, in a depfold only one set of accumulator functions compute with values in the result type of the depfold (*i.e.* pairs of values). The other set of accumulator functions compute a normal fold with one set of accumulator functions. The accumulator functions that compute over pairs of values returned from each set of functions are thus "dependent" on the computation done by the other set of "independent" accumulator functions.

The following example demonstrates a depfold for a datatype representing lambda calculus terms with some added features. We will use the depfold to insert "apply" functions and transform function values into data values when a function parameter is applied as a function. The datatype is as follows:

```
datatype Term =
    Var of string
  | Abs of string * Term
  | App of Term * Term
  | Let of string * Term * Term
  | Rep of Term
```

The constructors *Var*, *Abs*, and *App* are as one would expect. *Let* is used for local declarations. Using *Let*, a variable (the *string*) is assigned a value (the first *Term*) but that variable is declared only in the scope of the second *Term*. The *Rep* constructor is

used for the representation of a *Term* as a data value. This constructor will be used when a function value needs to be transformed into a data value.

The depfold for this datatype, $depfold^{Term}$, is as follows. The function $fold^{Term}$ is a normal fold over *Term*s.

$$
\begin{array}{l}
\textbf{fun } depfold^{Term}\ (\ (f \textbf{ as } (f_{Var}, f_{Abs}, f_{App}, f_{Let}, f_{Rep})),\\
\qquad\qquad\qquad (g \textbf{ as } (g_{Var}, g_{Abs}, g_{App}, g_{Let}, g_{Rep}))\ t\ =\\
\quad \textbf{case } t \textbf{ of}\\
\qquad Var\ a \quad\ \Rightarrow\ (\ f_{Var}\ a,\\
\qquad\qquad\qquad\ \ g_{Var}\ a)\\
\qquad |\ Abs\ (a,t) \ \Rightarrow\ (\ f_{Abs}\ (a, depfold^{Term}\ (f,g)\ t),\\
\qquad\qquad\qquad\ \ g_{Abs}\ (a, fold^{Term}\ g\ t))\\
\qquad |\ App\ (t_1,t_2) \ \Rightarrow\ (\ f_{App}\ (depfold^{Term}\ (f,g)\ t_1, depfold^{Term}\ (f,g)\ t_2),\\
\qquad\qquad\qquad\ \ g_{App}\ (fold^{Term}\ g\ t_1, fold^{Term}\ g\ t2))\\
\qquad |\ Let\ (a,d,b) \ \Rightarrow\ (\ f_{Let}\ (a, depfold^{Term}\ (f,g)\ d, depfold^{Term}\ (f,g)\ b),\\
\qquad\qquad\qquad\ \ g_{Let}\ (a, fold^{Term}\ g\ d, fold^{Term}\ g\ b))\\
\qquad |\ Rep\ t \quad\ \Rightarrow\ (\ f_{Rep}\ (depfold^{Term}\ (f,g)\ t),\\
\qquad\qquad\qquad\ \ g_{Rep}\ (fold^{Term}\ g\ t))
\end{array}
$$

The type of $depfold^{Term}$, without any accumulator function types except those for the *Abs* accumulator functions, is as follows:

$$
\begin{array}{l}
(\ldots * (string * (\alpha * \beta) \rightarrow \alpha) * \ldots) * (\ldots * (string * \beta \rightarrow \beta) * \ldots)\\
\rightarrow Term \rightarrow \alpha * \beta
\end{array}
$$

The types of the accumulator functions illustrate the dependency of one set of accumulator functions on the other set. Types of dependent accumulators have the type $(\alpha * \beta)$ in place of recursive references to the datatype, since α is the return type of the dependent accumulators and β is the return type of the independent accumulators. Types of independent accumulators have just the type β in place of recursive references to the datatype.

We will use this depfold to perform a simple program transformation on *Term*s. The independent portion of the depfold will calculate environment values for *Term*s, similar to the environment calculations performed in *Firstify*. An environment value for a *Term*

is either *Param*, indicating that the *Term* is a parameter, or *Unknown*. The dependent portion of the depfold will transform *App* constructors when the environment value of the first *Term* in the *App* is *Param*. The transformation converts the *App* into an application of the function "apply" to the two *Terms* in the *App*, and applies the *Rep* constructor to the first *Term* in the *App* (indicating that the function value is transformed into a data value). For example, if the environment value of *Var "p"* is *Param*, then *App*(*Var "p"*, *Var "x"*) is transformed into *App*(*App*(*Var "apply"*, *Rep*(*Var "p"*)), *Var "x"*).

The function *insert_apply*, shown in Figure 3.1, performs this transformation. The *env* parameter is the environment, which is a function that takes a string and returns either *Param* or *Unknown*. The dependent accumulator function for the *App* constructor performs the transformation. Each of the independent accumulator functions return an environment value. The independent accumulator function for the *Var* constructor computes an environment value by applying *env* to the string. The environment value returned by the independent accumulator function for the *Let* constructor is the environment value of the body (*i.e.* the second *Term*) of the *Let*.

This function does not work entirely as desired. In particular, environment information is not updated and propagated when environment bindings are created by *Abs* and *Let* constructors. This problem is addressed by introducing suspended computation, as described in the next Section.

Clearly this function could be coded more simply (and lucidly) with a normal fold. However, the depfold in this example demonstrates (a) the clear separation of the "transformation" functionality from the "environment computing" functionality; and (b) the dependence of the "transformation" functionality on the "environment computing" functionality.

A mutually recursive version of a depfold over *ereps* and *dreps*, similar to the one shown in the example, is used to perform transformations in *Firstify*. Accumulator functions for performing transformations are used for the dependent portion of the depfold, and accumulator functions for calculating environment values are used for the independent portion. Again, the functionality of this depfold could be obtained using a normal

```
fun insert_apply t env =
    let val f = ( (* Var *) Var,
                  (* Abs *) λ (a, (t, t_env)).Abs (a, t),
                  (* App *) λ ((t_1, t_1^env), (t_2, t_2^env)).
                               if t_1^env = Param then
                                   App (App (Var "apply", Rep t_1), t_2)
                               else
                                   App (t_1, t_2),
                  (* Let *) λ (a, (d, d_env), (b, b_env)).Let (a, d, b),
                  (* Rep *) λ (t, t_env).Rep t)
        val g = ( (* Var *) env,
                  (* Abs *) λ _.Unknown,
                  (* App *) λ _.Unknown,
                  (* Let *) λ (a, d, b).b,
                  (* Rep *) λ _.Unknown)
        val (t', t'_env) = depfold^Term (f, g) t
    in
        t'
    end
```

Figure 3.1: A simple program transformation on *Terms*

fold, but the depfold allows for a clean separation of environment computation from transformation computation.

3.2.2 Propagating Environment Information

One of the characteristics of fold-like functions is that information is propagated one direction only. In particular, if a value in a datatype is viewed as a tree, then information is propagated from the leaves to the root, but not in the opposite direction. But the transformation depfolds need to propagate environment information in both directions. Environments need to be propagated down to subexpressions, such as in a **let** expression, where the environment bindings created by the declarations need to be in the environment used in transforming the **let** body. To augment the environment with bindings from the declaration, the bindings need to propagate upward, but this is the natural propagation direction for fold-like functions.

Propagating environment information downwards is accomplished in the transformation depfolds via suspended computation. The value returned by a transformation depfold over *ereps* is $(environ \rightarrow erep * environ \rightarrow enventry)$ rather than $(erep * enventry)$ as one might expect. Thus, no value is computed until the environment in which the value exists is available.

Propagation of environment information in a fold function is similar to the propagation of state information via a *state transformer monad.* However, we need to have finer control over the way in which state information is propagated than the control possible with a state transformer monad. With the monad, the state is threaded in a particular direction. For example, with left to right threading the state available for a particular subterm computation is the state after computing over subterms to the left of the current subterm. This is too restrictive for propagating environment information, since environment values need sometimes to propagate left to right, sometimes right to left, and sometimes "in parallel" over subterms. An example of "parallel" threading is when computing over the values in a tuple. Each computation requires environment information, but no computation requires environment information for other values in

the tuple.

We will now modify the *Term* example to propagate environment information downwards by introducing suspended computations. A new version of the transformation function, *insert_apply'*, is shown in Figure 3.2. Each accumulator function now abstracts over the current environment (*currenv*) to introduce suspended computation. In the independent (environment value computing) accumulator functions, the current environment is used in determining environment values. The dependent accumulator functions also use the current environment to determine values of sub-*Term*s. Also note that in the dependent accumulator functions for the *Abs* and *Let* constructors, the environment used for determining the values of some sub-*Term*s is augmented with bindings introduced in the *Term*. In the *Abs* dependent accumulator function, the variable bound in the abstraction (attached to the environment value *Param*) is added to the environment used to process the body of the abstraction. In the *Let* dependent accumulator function, the variable defined in the *Let* (attached to the environment value of its corresponding *Term*) is added to the environment used to process the body of the *Let*. This environment augmentation is done with the function *augment*.

3.2.3 Understanding Transformation Code Samples

As stated above, most of the transformations performed by *Firstify* are coded as a set of mutually recursive depfolds over *ereps* and *dreps*. Thus, the most significant portions of the code that perform transformations are accumulator functions for the dependent part of a depfold over *ereps* or *dreps*. Most code samples in the remainder of this thesis are such accumulator functions. The type of one of these functions is

$$DOMAIN \rightarrow environ \rightarrow REP$$

where *REP* is either *erep* or *drep*, and *DOMAIN* is the domain of the constructor modified so that each *erep* in the domain is replaced with

$$environ \rightarrow erep * environ \rightarrow enventry$$

```
fun insert_apply' t env =
    let fun augment env (s, e) = λ x.if x = s then e else env x
        val f = ( (* Var *) λ a.λ currenv.Var a,
                  (* Abs *) λ (a, (t, t_env)).λ currenv.
                              Abs (a, t (augment currenv (a, Param))),
                  (* App *) λ ((t1, t1^env), (t2, t2^env)).λ currenv.
                              if t1^env = Param then
                                  App (App (Var "apply", Rep (t1 currenv)), (t2 currenv))
                              else
                                  App ((t1 currenv), (t2 currenv)),
                  (* Let *) λ (a, (d, d_env), (b, b_env)).λ currenv.
                              Let (a, d currenv, b (augment currenv (a, d_env currenv))),
                  (* Rep *) λ (t, t_env).λ currenv.Rep (t currenv))
        val g = ( (* Var *) currenv a,
                  (* Abs *) λ _.λ currenv.Unknown,
                  (* App *) λ _.λ currenv.Unknown,
                  (* Let *) λ (a, d, b).b currenv,
                  (* Rep *) λ _.λ currenv.Unknown)
        val (t', t'_env) = depfold^Term (f, g) t
    in
        t' env
    end
```

Figure 3.2: Modified *Term* program transformation

and each *drep* in the domain is replaced with

$$environ \rightarrow drep * environ \rightarrow (string * enventry)\ list$$

The meanings of function and variable values used by an accumulator function are indicated in a table below the code sample. The most commonly used function and variable values are described in Appendix B.

3.3 Execution Phases

Firstify transforms a source program in a series of four phases, described below. The series is repeated for each HOF set, until no more HOFs are found to be transformed. Phases II and III are repeated within the series, once for each HOF in the HOF set being transformed. A diagram illustrating phase execution order can be found in Figure 3.3.

All phases are executed by making a pass over the input program, and processing declarations as required by the phase. A common set of functions, parameterized on phase information, is used to execute all phases.

3.3.1 Phase I—Searching for a HOF

This phase is executed by looking for a parameter value applied as a function (a FVP). As soon as a FVP that has not already been detected is found, the FVP is recorded in the *transformation list* and Phase I ends. The transformation list is stored in a **ref** variable. If more than one FVP is detected simultaneously (such as when a **case** statement with several functional values is applied to an argument) only one FVP is recorded, as the others will be detected and recorded in Phase II. No program transformations are performed in this phase. Information recorded in the transformation list is used in the subsequent phases.

FVPs are encoded as values in the type *fparm*. Definitions of this type and related types follow.

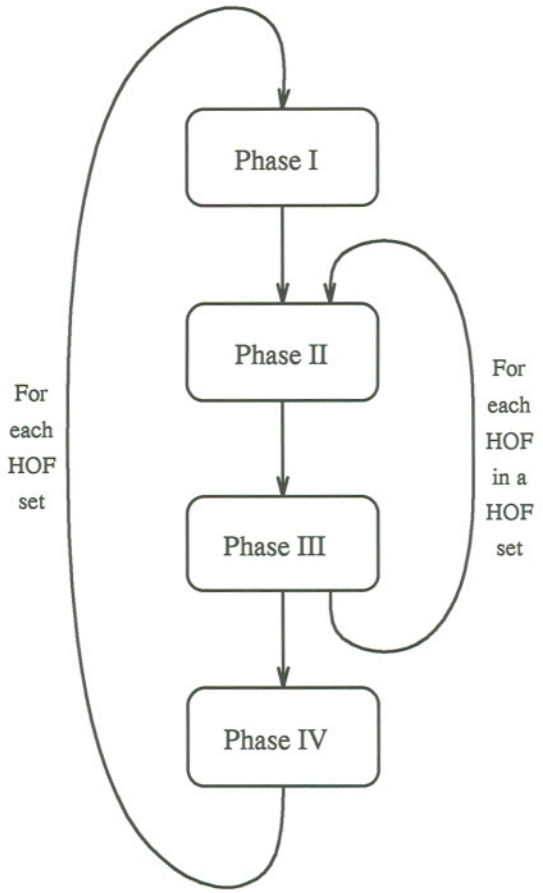

Figure 3.3: *Firstify* phase execution order

```
λ(fct, arg). λenv. let   val fct_env      =  lookup_enventry env ((second fct) env)
                         val side_effect  =  if refs_param fct_env then
                                                 add_new_xform (get_param_info fct_env)
                                             else
                                                 ()
                   in
                         App((first fct) env, (first arg) env)
                   end
```

Context	
Identifier	**Type**
	Description
add_new_xform	*fparm list* → ()
	Adds one FVP from the *fparm list* to the transformation list.

Figure 3.4: Phase I *App* accumulator function

```
type function_id        = string * int
datatype parameter_id   = UnknownPosition
                        | Position of int
                        | NestedAt of int * parameter_id
type fparm              = function_id * parameter_id
```

The *string* in *function_id* is the name of the function, and the *int* is the scope level at which the function is defined. Examples will omit the scope level when it is insignificant, representing functions simply by the function name.

The *parameter_id* datatype encodes the position of a value in a tuple. If a function's parameter is a singleton rather than a tuple, it is encoded as *Position* 0. *NestedAt* encodes positions that are within nested tuples. For example, the position of the **b** in the tuple **(a,((b,c),d))** would be encoded as *NestedAt* (2, *NestedAt* (1, *Position* 1)).

Figure 3.4 shows the accumulator function for the *App* constructor that is used in Phase I. In this function, FVPs are detected and recorded in the transformation list when the function being applied is a parameter value to some surrounding function.

```
λ (fct, arg).λ env.
    let val fct_env = lookup_enventry env ((second fct) env)
        val fct_exp = (first fct) env
        val arg_exp = (first arg) env
    in
        if refs_param fct_env then
            if member fvp (get_param_info fct_env) then
                App(make_id "apply_" base_fvp "", Tuple [fct_exp, arg_exp])
            else
                App(fct_exp, arg_exp)
        else
            App(fct_exp, arg_exp)
    end
```

Figure 3.5: Phase II *App* accumulator function

3.3.2 Phase II—Transforming a HOF

The body of a HOF is transformed by changing calls to the FVP into calls to the appropriate apply function for the FVP. Also, multiple alternate expressions are transformed if one alternate evaluates to the FVP, as described in Section 3.7.

Figure 3.5 shows the accumulator function for the *App* constructor that is used in Phase II. In this function, apply functions are inserted into the input program when the function being applied is the FVP currently being processed.

3.3.3 Phase III—Transforming Calls to a HOF

Any declaration that contains a call to the HOF is transformed in this phase, by transforming the function value passed to the HOF into a value in the FVP datatype. The HOF is also transformed (specialized) by tagging the function name based on the type of the FVP, as discussed in Chapter 5.

Complications to these transformations arising from **let** expressions are discussed in Chapter 4; representing functions in the FVP datatype is discussed in Section 3.6, and

HOF and FVP expressions with multiple alternates are discussed in Section 3.7.

As FVPs are transformed, information about them is recorded in the *apply information list*, used to construct the FVP datatype and apply function used to interpret FVP representations. The apply information list is stored in a **ref** variable. This information is used in Phase IV.

Figure 3.6 shows the accumulator function for the *App* constructor that is used in Phase III. This function turns a FVP expression into a value in the FVP datatype via the *modify_fvp* function. *Modify_fvp* both converts a FVP expression and records information about it in the apply information list.

3.3.4 Phase IV—Specializing HOFs and Adding Apply Functions

All HOF declarations in a HOF set are transformed in this phase, by creating multiple copies of the HOF declarations, one copy for each specialization group. Specialization is discussed in detail in Chapter 5.

Also in this phase, an apply function and FVP datatype are added to the source program. In fact, several sets of apply functions and FVP datatypes might be added due to specialization. When these declarations are added, the source program is reordered, as discussed in Section 3.8.

3.4 Detecting HOFs with Shared FVPs

In Phase I, information about HOFs is added to the transformation list. In order to execute all necessary transformations, other HOFs sharing FVPs with the HOF currently being processed need also to be included in the transformation list. HOFs with shared FVPs are detected in Phases II and III, and added to the transformation list during those phases.

In Phase II, shared FVPs can be detected in HOF declarations in two ways. First, if the FVP is used as an argument to another function, then the other function shares a FVP with the surrounding HOF. In the example below, when executing Phase II on the

```
λ (fct, arg).λ env.
    let val fct_env   =  lookup_enventry env ((second fct) env)
        val fct_exp   =  (first fct) env
        val arg_exp   =  (first arg) env
        val fvp_pos   =  second fvp
        val flist     =  get_function_info fct_env env
    in
        if member (first fvp) flist then
            let val actual_fvp  =  extract_param_exp arg_exp fvp_pos
                val actual_fvp_rep  =  modify_fvp actual_fvp fvp base_fvp scope env
            in
                App(fct_exp, insert_param_exp actual_fvp_rep arg_exp fvp_pos)
            end
        else
            App(fct_exp, arg_exp)
    end
```

Context	
Identifier	**Type**
	Description
modify_fvp	*string erep* → *fparm* → *fparm* → *int* → *environ* → *string erep* Turns a function name or lambda expression into a expression that represents the function. The first *fparm* is the FVP currently being transformed, and the second *fparm* is the base FVP. The *int* is the scope level at which to update the apply information list, which is done as a side effect.

Figure 3.6: Phase III *App* accumulator function

FVP (*high, Position 1*), the shared FVP (*high', Position 2*) is detected.

$$\textbf{fun}\ high\ (f,x)\ =\ (high'\ (x,f),\ f\ x)$$

Figure 3.7 shows the accumulator function for the *App* constructor used in Phase II, enhanced to include detection of shared FVPs as described. Lines which were added to this function that were not in Figure 3.5 are marked with *add* ▷.

Second, if the FVP is one alternate in a multiple alternate expression, and another parameter (to the HOF or some other surrounding function) is another alternate of the expression, then the other parameter is a FVP, shared with the current FVP. In the example below, when executing Phase II on the FVP (*high, Position 1*), the shared FVP (*high, Position 2*) is detected.

$$\begin{aligned}\textbf{fun}\ high\ (f,g,x)\ =\ (&\textbf{case}\ x\ \textbf{of}\\ &\quad nil\ \Rightarrow\ f\\ &\mid\ x :: xs\ \Rightarrow\ g)\ x\end{aligned}$$

Figure 3.8 shows the accumulator function for the *Case* constructor used in Phase II, that detects shared FVPs as described. Note that no transformations are performed by this function.

In Phase III, shared FVPs can be detected at HOF call sites in two ways. First, if the FVP being passed to the HOF is a parameter to a surrounding function, then the surrounding function shares a FVP with the current HOF. In the example below, when executing Phase III on the FVP (*high, Position 1*), the shared FVP (*outer, Position 1*) is detected.

$$\textbf{fun}\ outer\ (f,x)\ =\ high\ (f,x)$$

Figure 3.9 shows the accumulator function for the *App* constructor used in Phase III, enhanced to include detection of shared FVPs as described. Lines which were added to this function that were not in Figure 3.6 are marked with *add* ▷. Note that if the FVP is shared, the FVP expression is not transformed.

Second, if the HOF is one alternate of a multiple alternate expression, and a different HOF is another alternate of the expression, this HOF has a shared FVP with the current

```
    λ (fct, arg).λ env.
        let val fct_env = lookup_enventry env ((second fct) env)
            val fct_exp = (first fct) env
            val arg_exp = (first arg) env
add ▷       val arg_env = lookup_enventry env ((second arg) env)
add ▷       val side_effect =
add ▷           if refs_param arg_env then
add ▷               map (λ (Param fvp', p). if fvp' = fvp then
add ▷                                           add_xform_fun base_fvp [(exp_to_fid fct, p)]
add ▷                                       else
add ▷                                           ()
add ▷                       | (_, p).())
add ▷               (values_in arg_env)
add ▷           else
add ▷               nil
        in
            if refs_param fct_env then
                if member fvp (get_param_info fct_env) then
                    App(make_id "apply_" base_fvp "", Tuple[fct_exp, arg_exp])
                else
                    App(fct_exp, arg_exp)
            else
                App(fct_exp, arg_exp)
        end
```

Context	
Identifier	**Type**
	Description
exp_to_fid	*string erep* → *function_id*
	Converts an expression representing a function into a *function_id*.
values_in	*string erep* → (*string erep* * *parameter_id*) *list*
	Returns a list of the expressions and their corresponding positions contained in a tuple expression.

Figure 3.7: Phase II *App* accumulator function, modified to detect shared FVPs

```
λ (ce, pelist). λ env.
    let val p_exp_list =
            map (λ (p, (e_exp, e_env)).(p, e_exp (enrich env p (e_env env)))) pelist
        val exp_list = map second p_exp_list
        val env_list = map (λ (p, e).(second e) env) pelist
        val side_effect =
            case fparm_in fvp exp_list of
                just pos ⇒
                    let val shared_list = mult(map get_embedded_param_info env_list)
                        val shared_list' = filter (λ (fparm, pos').pos = pos') shared_list
                    in
                        add_xform_fun base_fvp (map first shared_list')
                    end
              | nothing ⇒ ()
    in
        Case((first ce) env, p_exp_list)
    end
```

Context	
Identifier	**Type**
	Description
fparm_in	*fparm* → *string erep list* → *parameter_id maybe* If the *fparm* value is found in any of the expressions in the list, then returns *just* the position of the *fparm* value in the expression. Otherwise returns *nothing*.
get_embedded_param_info	*enventry* → (*fparm* ∗ *parameter_id*) *list* Returns a list of all parameters referred to in an environment value, paired with their corresponding positions in the expression.

Figure 3.8: Phase II *Case* accumulator function

```
    λ (fct, arg).λ env.
        let val fct_env   =  lookup_enventry env ((second fct) env)
            val fct_exp   =  (first fct) env
add ▷       val arg_env   =  lookup_enventry env ((second arg) env)
            val arg_exp   =  (first arg) env
            val fvp_pos   =  second fvp
            val flist     =  get_function_info fct_env env
        in
            if member (first fvp) flist then
                let val actual_fvp  =  extract_param_exp arg_exp fvp_pos
add ▷               val actual_fvp_env  =  extract_param_env arg_env fvp_pos
                    val actual_fvp_rep  =  modify_fvp actual_fvp fvp base_fvp scope env
                in
add ▷               if refs_param actual_fvp_env then
add ▷                   (add_xform_fun base_fvp (get_param_info actual_fvp_env);
add ▷                    App(fct_exp, arg_exp))
                    else
                        App( fct_exp,
                             insert_param_exp actual_fvp_rep arg_exp fvp_pos)
                end
            else
                App(fct_exp, arg_exp)
        end
```

Figure 3.9: Phase III *App* accumulator function, modified to detect shared FVPs

FVP. In the example below, when executing Phase III on the FVP (*high, Position 1*), the shared FVP(*high', Position 1*) is detected.

```
fun call_high x = (case x of
                     nil  ⇒ high
                   | x :: xs ⇒ high') (f, x)
```

Figure 3.10 shows the accumulator function for the *Case* constructor used in Phase III, that detects shared FVPs as described. Note that no transformations are performed by this function.

3.5 New Identifiers Introduced by Transformations

Firstify introduces new declarations to the source program in the form of apply functions and FVP datatypes. The names of all identifiers declared in these declarations are constructed using the base FVP in the HOF set to which a HOF belongs.

Apply function names are all of the form **apply**_*position*_**of**_*function-name*_*tags*

Datatype names are all of the form **type**_*position*_**of**_*function-name*_*tags*

Constructor names are all of the form *fvp-id*_*position*_**of**_*function-name*_*tags*

where *fvp-id* is either the string "P_" followed by the name of the function being passed, or "P_Section$n$", where n is a number, if the function being passed is a lambda abstraction.

position is an encoding of the position of the base FVP in the parameter of the base HOF as a sequence of integers separated by underscores.

function-name is the name of the base HOF.

tags are one or more uppercase letters separated by underscores. Tags are introduced by specializing a HOF. Tags are omitted in examples shown in this thesis until Chapter 5.

Currently, no provision is made in the implementation to ensure that these identifiers are not already declared in the source program. Thus, it is assumed that the input program does not declare identifiers that conflict with those created by *Firstify*.

```
λ (ce, pelist). λ env.
    let val p_exp_list =
            map (λ (p, (e_exp, e_env)).(p, e_exp (enrich env p (e_env env)))) pelist
        val exp_list = map second p_exp_list
        val env_list = map (λ (p, e).(second e) env) pelist
        val side_effect =
            case function_in (first fvp) exp_list of
                just pos ⇒
                    let val shared_fct_list = mult(map get_embedded_fct_info env_list)
                        val shared_fct_list' = filter (λ (fid, pos').pos = pos') shared_fct_list
                        val shared_fvp_list =
                            map (λ (fid, _).(fid, (second fvp))) shared_fct_list'
                    in
                        add_xform_fun base_fvp (map first shared_fvp_list)
                    end
              | nothing ⇒ ()
    in
        Case((first ce) env, p_exp_list)
    end
```

Context	
Identifier	**Type**
	Description
function_in	*function_id* → *string erep list* → *parameter_id maybe* If the *function_id* is found in any of the expressions in the list, then returns *just* the position of the function in the expression. Otherwise returns *nothing*.
get_embedded_fct_info	*enventry* → (*function_id* ∗ *parameter_id*) *list* Returns a list of all functions referred to in an environment value, paired with their corresponding positions in the expression.

Figure 3.10: Phase III *Case* accumulator function

3.6 Representing Functions and Interpreting the Representations

Functions passed as FVPs are assumed to be in one of two forms at the HOF call site—a simple identifier or a *section*. Sections are lambda abstractions of the following form[1]:

$$\lambda\, p.f(\ldots,\ \hat{p},\ \ldots)$$

Where f is a function, p is a pattern, and $\hat{p}$ is the expression equivalent of p.

Each function passed as a FVP is represented with a constructed value in a FVP datatype. Each identifier passed as a FVP is represented by a nullary constructor which merely encodes the name of the function. For example, consider the following HOF call site (where (*high*, *Position 1*) is the FVP).

$$\textbf{val}\ z\ =\ high\ (f, x)$$

The identifier f would be represented by a nullary constructor, resulting in the following transformed declaration.

$$\textbf{val}\ z\ =\ high\ (P\_f\_1\_of\_high, x)$$

Each section passed as a FVP is represented by a constructor with the domain being a product of type variables, each type variable representing a variable in the section that is free in the context of the lambda abstraction but bound in the context of some surrounding function. The section is represented at the call site by applying the constructor to these variables. Variables that are free in the section but *not* bound in the context of a surrounding function are defined at a more outer scope than the surrounding function(s), and thus do not need to be represented in the datatype. For an example, consider the following HOF call site.

$$\textbf{fun}\ call\_high\ x\ =\ high\ (\lambda z.g(x, y, z), 0)$$

[1] According to the definition, abstractions in Standard ML can have multiple patterns, each with a corresponding expression. Pattern matching is used to select an expression to execute. This feature is not supported by *Firstify*.

In the section, x and y are free variables. However, only x is bound in the context of *call_high*. Thus, the only value that needs to be a constructor argument is x. The section would be represented by a constructor with argument x, as follows.

$$\textbf{fun } \mathit{call\_high}\ x\ =\ \mathit{high}\ (\mathit{P\_Section1\_1\_of\_high}\ x, 0)$$

The FVP datatype for representing the two calls to *high* presented above would need to be parameterized on a single type variable that represents the type of x in the section FVP. This datatype declaration is as follows.

$$\begin{array}{l}\textbf{datatype } \alpha\ \mathit{type\_1\_of\_high}\ = \\ \quad \mathit{P\_f\_1\_of\_high} \\ \mid\ \mathit{P\_Section1\_1\_of\_high}\ \textbf{of}\ \alpha\end{array}$$

In general, in a FVP datatype a type variable is needed for each constructor argument. However, suppose datatype D has a constructor C with a domain $(T_1 * \ldots * T_n)$ where T_i is D for some i. In this case, D is not parameterized with a type variable for T_i. Instead, the FVP datatype needs to be recursive. A recursive datatype is necessary if the following two conditions are met.

1. A section FVP is passed in *Position n* in a recursive call to a HOF.

2. The formal parameter to the HOF in *Position n* occurs freely in the section.

For example, consider the following program fragment, containing two calls to the HOF *map_with_policy*.

$$\begin{array}{l}\textbf{fun } \mathit{Policy}\ (f,x)\ =\ \textbf{if } x = 0\ \textbf{then } 0\ \textbf{else } f\ x \\ \textbf{fun } \mathit{map\_with\_policy}\ (f, nil) \quad\ = \ nil \\ \quad \mid\ \mathit{map\_with\_policy}\ (f, x :: xs)\ =\ (f\ x) :: (\mathit{map\_with\_policy}(\lambda y.\mathit{Policy}(f,y), xs)) \\ \textbf{val } \mathit{call\_mwp}\ =\ \mathit{map\_with\_policy}\ (foo, l)\end{array}$$

The first argument to *map_with_policy* (*i.e.* the argument in *Position 1*) is a FVP. In the recursive call to *map_with_policy*, the section $\lambda y.Policy(f,y)$ is passed as this FVP. Since f is a free variable in the section but is bound in the context of a surrounding

function (*map_with_policy*), it needs to be an argument to the constructor representing the section. Suppose we use a type variable for the type of this constructor argument in the datatype declaration, which results in the following declaration.

datatype α *type_1_of_map_with_policy* =
    P_Section1_1_of_map_with_policy **of** α
  | *P_foo_1_of_map_with_policy*

Since f (in the declaration of *map_with_policy*) is of type *type_1_of_map_with_policy*, the datatype needs to be instantiated with α as α *type_1_of_map_with_policy*. This instantiation is not possible in ML. Thus, the datatype should not be parameterized, and instead the domain of *P_Section1_1_of_map_with_policy* should be *type_1_of_map_with_policy*. This datatype is recursive.

datatype *type_1_of_map_with_policy* =
    P_Section1_1_of_map_with_policy **of** *type_1_of_map_with_policy*
  | *P_foo_1_of_map_with_policy*

Apply functions for a HOF set have two parameters—the representation of a function (in the FVP datatype) and the argument to that function. The apply function for a HOF set interprets function representations to apply the appropriate function to the argument. Interpreting representations of function identifiers is trivial. Representations of sections are interpreted by evaluating the body of the section, modified as follows:

- Each variable value stored in the constructor representing the section is substituted into the section body as appropriate.
- The argument to the represented function is substituted into the section body as appropriate.

The apply functions for the two examples above are:

```
fun apply_1_of_high (function_rep, function_arg) =
    case function_rep of
        P_f_1_of_high ⇒ function_arg
      | P_Section1_1_of_high x ⇒ g(x, y, function_arg)

fun apply_1_of_map_with_policy (function_rep, function_arg) =
    case function_rep of
        P_Section1_1_of_map_with_policy f ⇒ Policy(f, function_arg)
      | P_foo_1_of_map_with_policy ⇒ foo function_arg
and Policy (f, x) = if x = 0 then 0 else apply_1_of_map_with_policy (f, x)
```

Note that *apply_1_of_map_with_policy* now must be mutually recursive with *Policy*. Introduction of mutual recursion is described in Section 3.8.

3.7 Multiple Alternate Expressions

Multiple alternate expressions contain several explicit possible values, or *alternates*. It is not possible to statically determine which of the alternates will be selected when the expression is evaluated. Thus, transformations done by *Firstify* that are performed on one alternate of a multiple alternate expression will require transformations on the other alternates. The alternates must have unifiable types. Examples of multiple alternate expressions in ML are **case**, **handle**, and **if** ... **then** statements, and applications of lambda abstractions that have more than one pattern.

Since *Firstify* transforms expressions that evaluate to HOFs and FVPs, it must be capable of transforming multiple alternate expressions where one of the alternates contains a HOF or a FVP. In particular, *Firstify* must transform multiple alternate expressions with alternates containing one of the following values:

1. A FVP, when in the body of a HOF.
2. A FVP, at the call site of a HOF.
3. A HOF, at the call site of a HOF.

Firstify transforms **case** and **handle** expressions in these three contexts. **If ... then** statements are represented in the CRML abstract syntax as **case** statements, obviating the need to deal with them explicitly. Also, since it is assumed that the only lambda abstractions in the input to *Firstify* are sections that are passed as FVPs, lambda abstraction applications are not handled by *Firstify*. However, the transformations described in this chapter would work on **if ... then** statements and lambda abstraction applications if such transformations were required.

3.7.1 FVP Alternates in HOF Bodies

Multiple alternate expressions with FVP alternates in HOF function bodies affect the transformations performed by *Firstify* in two ways.

1. Shared FVPs are detected, as described in Section 3.4. Shared FVPs with HOFs declared at different scopes are discussed in Section 4.3.2.

2. Alternates of a multiple alternate FVP expression that are not parameters to some surrounding function need to be transformed, as described below.

Consider a multiple alternate expression with an alternate that evaluates to the FVP currently being transformed. If an alternate is not a parameter to a surrounding function (*i.e.* a shared FVP), it must be transformed into a function representation. For example, consider the following program fragment.

$$\begin{aligned}
&\textbf{fun } high\ (f, x)\ =\ (\textbf{case } x \textbf{ of} \\
&\qquad\qquad\qquad\qquad nil\ \Rightarrow\ id \\
&\qquad\qquad\qquad\quad |\quad x :: xs\ \Rightarrow\ f)\ x \\
&\textbf{val } z\ =\ high\ (foo, [1])
\end{aligned}$$

When the declarations are transformed so that a call to the appropriate apply function is inserted where f is called, the resulting declarations are:

$$\begin{aligned}
\textbf{fun}\ \mathit{high}\ (f,x)\ = \mathit{apply_1_of_high}\ \ ((\textbf{case}\ x\ \textbf{of}\ & \\
& \mathit{nil}\ \Rightarrow\ \mathit{id} \\
& \mid\ y :: ys\ \Rightarrow\ f), x) \\
\textbf{val}\ z\ =\ \mathit{high}\ (\mathit{P_foo_1_of_high}, [1]) &
\end{aligned}$$

In this version, f is no longer a function type. It now has the type *type_1_of_high*. For the **case** expression to be well typed, both alternates must have unifiable types. Thus *id* needs to be transformed into something of type *type_1_of_high*, as follows:

$$\begin{aligned}
\textbf{fun}\ \mathit{high}\ (f,x)\ = \mathit{apply_1_of_high}\ \ ((\textbf{case}\ x\ \textbf{of}\ & \\
& \mathit{nil}\ \Rightarrow\ \mathit{P_id_1_of_high} \\
& \mid\ y :: ys\ \Rightarrow\ f), x) \\
\textbf{val}\ z\ =\ \mathit{high}\ (\mathit{P_foo_1_of_high}, [1]) &
\end{aligned}$$

In this example, each alternate evaluates to a function value. In general, each alternate might evaluate to some data structure containing a function value. The data structure might include tuples, records, or constructed values in a datatype. *Firstify* will correctly transform function values contained in a tuple, but not function values that are part of a record or constructed datatype values. This applies to all transformations, not just those for multiple alternate expressions. Transforming function values contained in other data structures is discussed in Section 6.1.

3.7.2 FVP Alternates at HOF Call Sites

At a HOF call site, the FVP expression can have multiple alternates. This affects the transformations done by *Firstify* in two ways.

1. Functions passed as FVPs need to be transformed into representations of the functions. If the FVP expression has multiple alternates, then each alternate needs to be transformed. This is done by the *modify_fvp* function, described in Section 5.2.

2. All functions passed as FVPs in a multiple alternate FVP expression must be coerced into the same specialization group. See Chapter 5 for more information.

3.7.3 HOF Alternates at HOF Call Sites

At a HOF call site, the HOF expression can have multiple alternates. This affects the transformations done by *Firstify* in two ways.

1. Shared FVPs are detected, as described in Section 3.4. Alternates of the HOF expression containing HOFs declared at different scopes are discussed in Section 4.3.2.

2. Specialization requires that names of HOF functions be modified. If the HOF expression has multiple alternates, each alternate needs to be modified. See Chapter 5 for more information.

3.8 Ordering Declarations

When declarations are added to the input program, they must be added so that declarations referring to an identifier occur after the declaration of that identifier. *Firstify* ensures this by reordering declarations each time a set of FVP datatype and apply function declarations are added to a program.

The algorithm operates by constructing and manipulating a directed graph containing declaration information. Each node contains one or more mutually recursive declarations and a unique identification number. Arcs indicate dependencies between one declaration and another. When a declaration D_1 references an identifier declared in another declaration D_2, there is an arc from D_1 to D_2.

After the initial graph is built from the current program (list of declarations) and declarations to be inserted, cycles are detected and eliminated. Cycles are detected using a depth first search, which keeps tracks of nodes visited and returns a list of node numbers found in a cycle. Cycles are eliminated by combining the nodes in a cycle into one node and adjusting arcs pointing to any node in the cycle to point to the one node.

When all cycles have been eliminated, a reordered program is constructed. This is done with a topological sort of the graph.

This algorithm requires unique naming of all identifiers declared at the same scope level. This is an input assumption for *Firstify*.

Chapter 4

Local Declarations

Modern functional languages often allow declarations that are local to another declaration. ML provides this feature with the **let** statement. A **let** statement has two parts: a **let** *declaration* and a **let** *body*. The identifiers declared in the **let** declaration are visible only in the **let** body.

The effects of local declarations on the transformations performed by *Firstify* are discussed in this chapter. Although it is assumed that the input to *Firstify* has all functions declared at the "top level", some of what is discussed in this chapter has been implemented in *Firstify*. In particular, the details discussed in Sections 4.1 and 4.2 have been implemented.

4.1 Scopes and Scope Levels

Each **let** declaration defines a new scope[1]. At runtime, value bindings for identifiers declared in the new scope supersede previous bindings for the same identifiers. Likewise, an environment binding created by *Firstify* is superseded by a new environment binding created where an identifier is redeclared at a new scope.

Scopes are encoded in *Firstify* as an integer called the *scope level*. The initial scope level is 0. Each time a **let** declaration is entered, the scope level is incremented. When the **let** body is exited, the scope level is decremented. Thus, lower scope levels refer

[1]Each **let** statement may in fact have several declarations. However, the CRML abstract syntax represents multiple declarations as nested **lets**, and so each declaration is really declared in a new scope.

to more "outer" scopes, and higher scope levels refer to more "inner" scopes. In the following example, x, *fact_sq*, and *fact_x_sq* are declared at scope level 0; *fact* is declared at scope level 1; *sq* is declared at scope level 2.

```
val x = 5

fun fact_sq n =
    let fun fact 0 = 1
          | fact n = n * (fact(n - 1))
    in
        let fun sq x = x * x
        in sq(fact n) end
    end

val fact_x_sq = fact_sq x
```

Scope levels are used in environment bindings to record the scope at which an identifier is defined. This information is recorded in the *LocalDec* constructor, and is used in performing transformations. Environment bindings for the identifiers declared in the above program are:

$$\begin{aligned} x &\xrightarrow{b} \mathit{LocalDec}(0, \text{“}x\text{”}) \\ \mathit{fact\_sq} &\xrightarrow{b} \mathit{LocalDec}(0, \text{“}\mathit{fact\_sq}\text{”}) \\ \mathit{fact} &\xrightarrow{b} \mathit{LocalDec}(1, \text{“}\mathit{fact}\text{”}) \\ \mathit{sq} &\xrightarrow{b} \mathit{LocalDec}(2, \text{“}\mathit{sq}\text{”}) \\ \mathit{fact\_x\_sq} &\xrightarrow{b} \mathit{LocalDec}(0, \text{“}\mathit{fact\_x\_sq}\text{”}) \end{aligned}$$

4.2 Transforming Local Declarations

Clearly, locally declared functions can be higher order, or can be used as FVPs. Thus, *Firstify* must be able to correctly transform locally declared HOFs, represent locally declared functions that are FVPs, and insert apply function and FVP datatype declarations at the appropriate scope level.

Identifiers declared at a particular scope are visible only at that scope and more "inner" scopes. Thus, HOFs declared at scope level n are visible only at scope levels

$\geq n$. As a result, transformations to a HOF and calls to the HOF only occur at the scope level of the HOF definition or greater scope levels. To ensure this, in Phase I *Firstify* performs "local defunctionalization" on each scope. Local defunctionalization is performed by transforming the declaration and body of the **let** statement into a complete program (*i.e.* a list of declarations), and calling the top level function of *Firstify* recursively on this program. The transformation and apply information lists used for a local defunctionalization are unique to that particular call to *Firstify*. Thus, only HOFs declared in the current "program" are transformed. Figure 4.1 shows the accumulator function for the *Letrec* constructor used in Phase I, which performs local defunctionalization as described.

Transforming a **let** statement into a list of declarations is a straightforward process of appending the **let** declaration with a declaration built by assigning the **let** body to some identifier not declared or referenced in the **let** statement.

Local defunctionalization ensures that the apply function and FVP datatype declarations created by *Firstify* are placed in the program at a scope level less than or equal to the scope level where the HOF is declared. This is the desired placement, since the HOF refers to these declarations.

4.3 Problems with Local Declarations

A declaration D_1 *precedes* another declaration D_2 if

- the scope level of D_1 is less than the scope level of D_2

  or

- the scope levels of the two declarations are equal and either
  - D_1 comes before D_2 in the list of declarations declared at that scope level

    or
  - D_1 and D_2 are mutually recursive

```
λ(letdec, letbody).λenv.
    let val letdec_env  = lookup_enventry env ((second letdec) env)
        val letdec_dec  = (first letdec) env
        val letbody_exp = (first letbody) (add_bindings letdec_body env)
        val new_ident   = unused_id (Letrec(letdec_dec, letbody_exp))
        fun rebuild_let nil body = body
          | rebuild_let ((dec as ⟨⟨val 'i = 'e⟩⟩) :: rest)) body =
                if i = new_ident then rebuild_let rest e
                else Letrec(dec, rebuild_let rest body)
          | rebuild_let (dec :: rest) body = Letrec(dec, rebuild_let rest body)
        val program     = letdec_dec :: ⟨⟨val 'new_ident = 'letbody_exp⟩⟩ :: nil
    in
        rebuild_let (run_firstify program env (scope + 1)) nil
    end
```

Context	
Identifier	**Type**
	Description
add_bindings	*(string * enventry)list → environ → environ*
	Add bindings to an environment, yielding a new environment.
run_firstify	*string drep list → environ → int → string drep list*
	Defunctionalize a program, using the given environment and scope level.
unused_id	*string erep → string*
	Return an identifier not declared or referenced in an expression.

Figure 4.1: Phase I *Letrec* accumulator function

For a transformed program to execute properly, the following restrictions on declarations are required:

1. Declarations of functions passed as FVPs must precede the apply function declarations for the HOF set.

2. An FVP datatype declaration must precede the corresponding apply function declaration.

3. Apply function declaration(s) must precede the corresponding HOF declaration(s).

4. A HOF declaration must precede all declarations containing calls to the HOF.

Restriction 4 is assumed to be met by the input program, and will be maintained by the the declaration reordering method used by *Firstify*, which is discussed in Section 3.8. Section 4.3.1 discusses the difficulties in meeting restriction 1. Section 4.3.2 discusses the difficulties in meeting restrictions 2 and 3.

4.3.1 HOF and FVP Declared at Different Scope Levels

From the four restrictions we can see that declarations of functions passed as FVPs must precede the declaration of the HOF to which the functions are passed. If the FVPs and HOF are all declared at the same scope level, this restriction can be met by the reordering method. However, if an FVP is declared at a scope level greater than the HOF declaration's scope level, another solution is needed.

Although the problem has not been thoroughly examined, the solution appears to be to represent the body of the function passed as the FVP in the same way sections are represented, as discussed in Section 3.6. Thus the apply function will directly execute the body of the function passed as a FVP rather than call that function. This technique could in fact be extended to *all* functions passed as FVPs, although a considerable amount of the input program's structure would be lost by such a transformation.

In the following program, *high* is a HOF declared at scope level 0, and *concat_to_x* is passed as a FVP to *high*, but is declared at scope level 1.

```
fun high (f,x) = f x

fun comma_separate (x,y) =
    let fun concat_to_x s = x ^ "," ^ s
    in high (concat_to_x,y) end
```

To transform this program, the body of *concat_to_x* needs to be executed in the apply function for *high*, and the identifier *concat_to_x* in the call to *high* must be replaced by a constructor representing the body of *concat_to_x*.

```
datatype α type_1_of_high =
    P_Lambda1_1_of_high of α

fun apply_1_of_high (function_rep, function_arg) =
    case function_rep of
        P_Lambda1_1_of_high x ⇒ x ^ "," ^ function_arg

fun high (f,x) = apply_1_of_high(f,x)

fun comma_separate (x,y) =
    let fun concat_to_x s = x ^ "," ^ s
    in high (P_Lambda1_1_of_high x,y) end
```

Note that in this example, the declaration of *concat_to_x* could be removed from the program, but of course this is not true in general.

It would pose some problems to include this solution in the current implementation of *Firstify*. All other transformations (with the exception of those in Phase IV that copy declarations and rename declared variables) can be performed at the level of an expression, where the only "state" information available is the environment, the transformation list, type information, and environment values of constituent parts of the expression. This transformation would require the propagation of function bodies to HOF call sites, which could be done by either (significantly) extending the state or adding an additional phase to the implementation.

4.3.2 Shared FVPs for HOFs Declared at Different Scope Levels

According to restrictions 2 and 3, the apply function and FVP datatype declarations for a FVP function must precede declarations of all HOFs which share that FVP. Recall that apply function and FVP datatype declarations are added to a program at the scope level of the base HOF, that is, the first HOF detected in a HOF set. Thus, the simplest way to ensure that restrictions 2 and 3 are met is to require the base HOF to be one of the "outermost" HOFs (*i.e.* a HOF declared at the least scope level of all HOF scope levels in the set).

However, *Firstify* detects HOFs in the order in which they are declared in the input program, and cannot guarantee that a base HOF is the outermost one in its HOF set. Also, since HOFs are detected only when a parameter is applied as a function, if the outermost HOF never applies its FVP, it might never be directly detected as a HOF[2]. For example, in the following program, *inner* shares a FVP with both *outer* and *high*. It would be desirable for either *outer* or *high* to be the base HOF. However, since *outer* never applies its FVP, and *inner* comes before *high* in the program *inner* will be detected first and will be the base HOF for this HOF set.

```
fun dec x = x - 1

fun outer (f,x) =
    let fun inner (g,x) = x * (outer(g,g x))
    in
        if x = 0 then 1
        else inner(f,x)
    end

fun high (f,n) = (f n, outer(f,n))

fun dec_and_fact n = high(dec,n)
```

This problem is compounded by local defunctionalization, because even if either *outer*

[2]If higher order functions were detected by type analysis, this would not be a problem

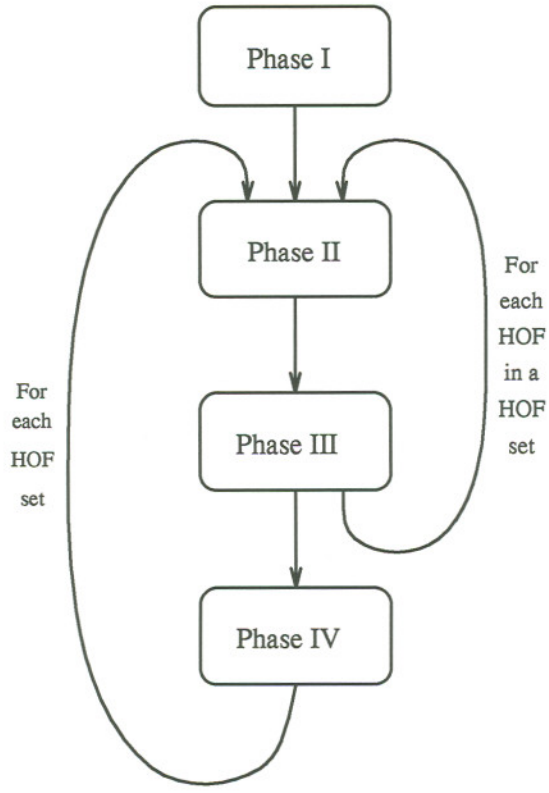

Figure 4.2: Phase execution order for first solution to local declarations problem

or *high* were the base HOF, local defunctionalization would cause the apply function and FVP datatype declarations for *inner* to be inserted at the same scope level as *inner*.

There appear to be two approaches to solve this problem. Both ensure that the base HOF for a HOF set is declared at the outermost scope.

The first approach is to modify phase execution order and Phase I of *Firstify*. In this approach, Phase I would find and record all HOFs at one time, creating a transformation list for each scope. Each transformation list would contain the HOF sets consisting of HOFs declared at that scope associated with the appropriate base HOF. The remaining phases will thus always use the correct base HOF in performing transformations. A diagram illustrating this modified phase execution order is shown in Figure 4.2.

This approach is "clean", and for the most part relies on code that is already developed, but would require substantial modifications to the *Firstify* implementation.

The second approach would be simpler to implement, but is perhaps less "elegant" than the first. Suppose H_{inner} is a HOF declared at scope level SL_{inner}, H_{outer} is a HOF declared at scope level SL_{outer}, and $SL_{inner} > SL_{outer}$. Suppose also that while performing transformations for H_{inner} it is discovered that H_{inner} shares a FVP with H_{outer}. At this point, H_{inner} could be recorded in the transformation list as a HOF with a FVP shared with the base HOF for H_{outer}. If transformations for H_{inner} had already been performed, then they would need to be re-performed with the appropriate base HOF. This would require a new phase (call it Phase IIIa), in which transformations for a particular base HOF could be "undone". This phase would execute as needed after Phase II or III. After executing this new phase, Phases II and III would be repeated with the correct base HOF. A diagram illustrating this modified phase execution order is shown in Figure 4.3.

Neither approach to solving this problem has yet been implemented. As stated above, it appears that the first approach is superior in terms of elegance, but the second approach would be simpler to implement.

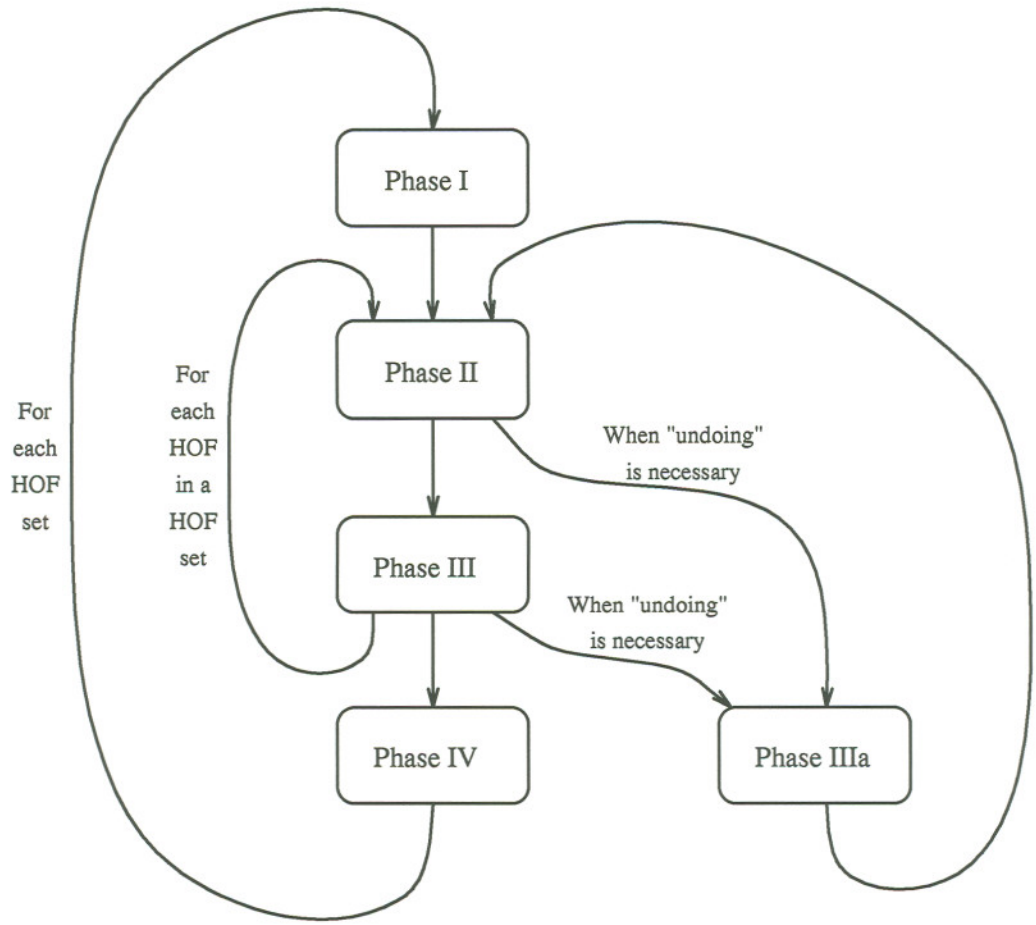

Figure 4.3: Phase execution order for second solution to local declarations problem

Chapter 5

Specialization

Consider the following program:

$$
\begin{array}{lcl}
\textbf{fun}\ \mathit{plus1}\ (x : \mathit{int}) & = & x + 1 \\
\textbf{fun}\ \mathit{java}\ s & = & s\ \hat{}\ \text{“}\mathit{java}\text{”} \\
\textbf{fun}\ \mathit{high}\ (f, x) & = & f\ x \\
\textbf{val}\ \mathit{two} & = & \mathit{high}\ (\mathit{plus1}, 1) \\
\textbf{val}\ \mathit{coffee} & = & \mathit{high}\ (\mathit{java}, \text{“}\mathit{mocca}\text{-”})
\end{array}
$$

According to Reynold's algorithm, a defunctionalized version of this program is:

$$
\begin{array}{l}
\textbf{fun}\ \mathit{plus1}\ (x : \mathit{int}) = x + 1 \\
\textbf{fun}\ \mathit{java}\ s = s\ \hat{}\ \text{“}\mathit{java}\text{”} \\
\textbf{datatype}\ \mathit{type\_1\_of\_high} = \\
\quad \mathit{P\_plus1\_1\_of\_high} \\
\quad \mid\ \mathit{P\_java\_1\_of\_high} \\
\textbf{fun}\ \mathit{apply\_1\_of\_high}\ (\mathit{function\_rep}, \mathit{function\_arg}) = \\
\quad \textbf{case}\ \mathit{function\_rep}\ \textbf{of} \\
\qquad \mathit{P\_plus1\_1\_of\_high} \Rightarrow \mathit{plus1}\ \mathit{function\_arg} \\
\qquad \mid\ \mathit{P\_java\_1\_of\_high} \Rightarrow \mathit{java}\ \mathit{function\_arg} \\
\textbf{fun}\ \mathit{high}\ (f, x) = \mathit{apply\_1\_of\_high}\ (f, x) \\
\textbf{val}\ \mathit{two} = \mathit{high}\ (\mathit{P\_plus1\_1\_of\_high}, 1) \\
\textbf{val}\ \mathit{coffee} = \mathit{high}\ (\mathit{P\_java\_1\_of\_high}, \text{“}\mathit{mocca}\text{-”})
\end{array}
$$

The function *apply_1_of_high* is not well typed. In particular, the function application *plus1 function_arg* implies that *function_arg* is an *int*, but the function application *java function_arg* implies that *function_arg* is a *string*. One way to fix this problem is

to create two versions of the apply function—one that handles *int* arguments, and one that handles *string* arguments. This would require two copies of *high*, each copy calling a different version of the apply function. The calls to *high* would then need to be modified to call the appropriate version of *high*. Finally, we should split the datatype into two datatypes—one for representations of functions of type $int \rightarrow int$ and another for representations of functions of type $string \rightarrow string$. The following program shows these changes. All declarations in this program are well typed.

```
fun plus1 (x : int)      = x + 1
fun java s               = s ^ "java"
datatype type_1_of_high_int =
    P_plus1_1_of_high_int
datatype type_1_of_high_string =
    P_java_1_of_high_string
fun apply_1_of_high_int (function_rep, function_arg) =
    case function_rep of
        P_plus1_1_of_high_int ⇒ plus1 function_arg
fun apply_1_of_high_string (function_rep, function_arg) =
    case function_rep of
        P_java_1_of_high_string ⇒ java function_arg
fun high_int (f, x)      = apply_1_of_high_int (f, x)
fun high_string (f, x)   = apply_1_of_high_string (f, x)
val two                  = high_int (P_plus1_1_of_high_int, 1)
val coffee               = high_string (P_java_1_of_high_string, "mocca-")
```

Creating multiple copies of HOFs and their associated apply functions and datatypes is known as *type specialization*, since a copy of each declaration is made especially for FVPs of a particular type. In general, *specialization* is a program transformation that creates specialized copies of a declaration based on some criterion. Higher order removal as described in Section 1.2 is an example of specialization, in this case based on the function values passed to a higher order function. Although specialization has this more general meaning, references to specialization in this thesis refer to type specialization.

In the above program, all identifiers that are added to the program by *Firstify*, as well as the names of the HOFs, are *tagged* with an additional string (or *tag*) which makes the identifier unique. In the actual *Firstify* implementation, single uppercase letters are used for tags rather than type names.

To specialize a program, the FVPs passed to any HOF in a HOF set are divided into *specialization groups*. Each specialization group requires its own unique tag, FVP datatype, apply function, and copies of the HOFs in the HOF set. Specialization groups are defined over the types of FVPs. Each group corresponds to a unique type. In the above example there are two specialization groups—one for FVPs of type $int \rightarrow int$, and the other for FVPs of type $string \rightarrow string$. The process of determining the specialization group for a FVP via type unification is discussed in Section 5.2.

5.1 The Specialization Transformations

Specialization entails the following transformations:

1. At the HOF call site, the specialization group and corresponding tag must be determined. The constructor name representing the FVP and HOF function name are tagged with this tag.

2. Tagged copies of all HOFs in a HOF set must be created.

3. Tagged copies of apply functions must be created.

4. Tagged copies of FVP datatypes must be created.

Transformation 1 is performed in Phase III. The other three are performed in Phase IV.

5.1.1 Specializing HOF Calls

At a HOF call site, the type of the function passed as a FVP determines the specialization group and corresponding tag. The HOF name and constructor used to represent the FVP must be tagged with this tag. Figure 5.1 shows the accumulator function for the *App*

constructor used in Phase III, enhanced to determine the specialization group and tag identifiers. Lines that were changed in this function from the version in Figure 3.9 are marked with *change* ▷. In this function, the constructor is tagged by calling *modify_fvp*, which now returns a pair consisting of the tagged constructor name representing the FVP and the tag for the specialization group. A partial declaration of *modify_fvp* is shown in Figure 5.2. This function handles only two kinds of FVP expressions—identifiers and sections. Multiple alternate FVP expressions are discussed in Section 5.2. Other kinds of FVP expressions are not included because of assumed restrictions on the input to *Firstify* (*e.g*, there are no **let** expressions).

In *modify_fvp*, the type of the FVP expression is determined via *get_fvp_type*, as discussed in Section 5.3. Information about the FVP is stored in the apply information list with either *store_fvp_info* or *store_section_fvp_info*. Finally, the constructor and tag are returned.

The tag returned by *modify_fvp* is passed to *add_tag*, which tags the HOF name. Figure 5.3 shows the definition of *add_tag*. As with *modify_fvp*, not all kinds of expressions are handled because of assumed restrictions on the input to *Firstify*.

5.1.2 Creating Specialized HOFs

In Phase IV, all HOFs in a HOF set need to be specialized. Specializing a HOF declaration involves the following transformations.

1. One copy of the HOF is created for each specialization group. The name of the HOF is tagged with the tag for the specialization group.

2. In the HOF body, references to names of other HOFs in the same HOF set are tagged with the tag for the specialization group.

3. In the HOF body, references to the apply function for the HOF set are tagged with the tag for the specialization group.

Transformation 1 is straightforward. Before each copy is inserted into the input program, transformations 2 and 3 are performed on the copy to tag identifiers. Figure 5.4

```
λ (fct, arg).λ env.
    let val fct_env   =  lookup_enventry env ((second fct) env)
        val fct_exp   =  (first fct) env
        val arg_env   =  lookup_enventry env ((second arg) env)
        val arg_exp   =  (first arg) env
        val fvp_pos   =  second fvp
        val flist     =  get_function_info fct_env env
    in
        if member (first fvp) flist then
            let val actual_fvp  =  extract_param_exp arg_exp fvp_pos
                val actual_fvp_env  =  extract_param_env arg_env fvp_pos
change ▷            val (actual_fvp_rep, group_tag)  =
                        modify_fvp actual_fvp fvp base_fvp scope env
            in
                if refs_param actual_fvp_env then
                    (add_xform_fun base_fvp (get_param_info actual_fvp_env);
                     App(fct_exp, arg_exp))
                else
change ▷                App( add_tag fct_exp group_tag env,
                             insert_param_exp actual_fvp_rep arg_exp fvp_pos)
            end
        else
            App(fct_exp, arg_exp)
    end
```

Figure 5.1: Phase III *App* accumulator function, modified to specialize HOF calls

```
fun modify_fvp fvp_expr fvp base_fvp scope env =
    let val fvp_type = get_fvp_type fvp_expr
    in
        case fvp_expr of
            Id nm ⇒
                let val id_to_use = deref_id a env
                    val tag = store_fvp_info base_fvp id_to_use fvp_type
                in
                    (Id (make_id ("P_" ^ id_to_use ^ "_") base_fvp tag), tag)
                end
          | Abs pelist ⇒
                let val vars_in_section = section_vars pelist env
                    val (tag, section_num) = store_section_fvp_info base_fvp vars_in_section
                                                                    pelist fvp_type env
                    val rep = make_id ("P_Section" ^ section_num ^ "_") base_fvp tag
                in
                    if vars_in_section = nil then
                        (Id rep, tag)
                    else
                        (App(Id rep, Tuple(map Id vars_in_section)), tag)
                end
    end
```

Context	
Identifier	**Type**
	Description
section_vars	*(string pattern * string erep) list → environ → string list* Returns a list of variables that are free in the body of a lambda abstraction but bound in the context of a surrounding function.
store_fvp_info	*fparm → string → string trep → string* Stores FVP information in the apply information list when the FVP expression is an identifier. The arguments, in order, are the base FVP, the name of the FVP function, and the type of the FVP function. Returns the tag for the FVP's specialization group.
store_section_fvp_info	*fparm → string list → (string pattern * string erep) list → string trep → environ → string* Stores FVP information in the apply information list when the FVP expression is a section. The arguments, in order, are the base FVP, a list of free variables used in the section, the body of the section, and the type of the section. Returns a pair consisting of the tag for the FVP's specialization group and a number uniquely identifying this section in the set of sections passed to the HOF.

Figure 5.2: *modify_fvp*

```
fun add_tag exp tag env =
    case exp of
        Id x                  ⇒ Id((deref_id x env) ^ "_" ^ tag)
      | Case(cexp, pelist)    ⇒ Case( cexp,
                                      map (λ(p, e).(p, add_tag e tag (augment env p cexp)))
                                          pelist)
      | Handle(exp, pelist)   ⇒ Handle( add_tag exp tag env,
                                        map (λ(p, e).(p, add_tag e tag env)) pelist)
      | x                     ⇒ x
```

Figure 5.3: *add_tag*

shows the accumulator function for the *Id* constructor used in Phase IV to perform this transformation. Note that this transformation is performed *only* on the declarations of functions in the HOF set, not on all declarations in the program.

5.1.3 Creating Specialized Apply Functions and FVP Datatypes

The apply information list contains all necessary information for constructing apply function and FVP datatype declarations. Each HOF set recorded in the apply information list yields a set of declarations—one apply function and FVP datatype for each specialization group for the HOF set. Each FVP recorded in the apply information list yields one constructor for the FVP datatype and one **case** alternate in the apply function.

5.2 Determining Specialization Groups

Specialization groups and the types associated with them are stored in the apply information list. Once the type of the FVP is determined (see section 5.3), the specialization group can be ascertained by systematically comparing the FVP type with the specialization group types already recorded in the apply information list.

```
λa.λenv. if member a hof_set_names then
             Id(a ^ "_" ^ tag)
         else if a = apply_name then
             Id(a ^ tag)
         else
             Id a
```

Context	
Identifier	**Type**
	Description
apply_name	*string*
	Apply function name (ends with "_").
hof_set_names	*string list*
	Names of functions in HOF set.

Figure 5.4: Phase IV *Id* accumulator function

5.2.1 Forced Specialization Groups

A complication to this method arises when several FVPs are *forced* to be in the same specialization group. For example, consider this program:

```
fun plus1 (x : int) = x + 1
fun id x            = x
fun high (f, x)     = f x
fun one_or_two b    = (case b of
                           true  ⇒ id
                         | false ⇒ plus1) 1
```

In the function *one_or_two*, both of the FVPs *id* and *plus1* must be in the same specialization group. If they are not, then the constructor values used to represent the two FVPs will be constructors for different datatypes, and thus the **case** expression will not be well typed.

To force all alternates in a multiple alternate FVP expression to be in the same specialization group, the *least general type* of all alternates in the multiple alternate expression is determined and used as the type of the FVP expression. The least general type is determined via type unification. The unification algorithm used for this is taken from [Pau91]. This algorithm either successfully unifies a list of type representations and returns a list of substitutions or fails. If the unification succeeds, the substitutions are performed on one of the type representations, yielding the least general type of the types. In this case we know that the unification will succeed, since all alternates in a multiple alternate expression must have unifiable types.

In the above example, the least general type of *id* and *plus1* is $int \rightarrow int$, and so *id* and *plus1* will both be in the specialization group corresponding to this type.

Figure 5.5 shows the definition of *modify_fvp'*, a version of *modify_fvp* enhanced to deal with multiple alternate expressions. Lines which were changed from the version in Figure 5.2 are marked with *change* ▷, and new lines are marked with *add* ▷. The only multiple alternate expression constructor shown is *Case*, although the computation for the *Handle* constructor is similar.

Note that *modify_fvp'* has an additional parameter of type *string trep maybe*. This parameter is used to force the FVP type to a particular value. It is only needed for recursive calls to *modify_fvp'*, so the call to *modify_fvp'* in Figure 5.1 should pass the value *nothing* as this parameter. Note also that the function *get_fvp_type* must be able to calculate the type of a multiple alternate FVP expression. This is discussed in Section 5.3.

5.2.2 FVPs in All Specialization Groups

In two instances, a FVP must be included in *all* specialization groups for the HOF set. The instances are:

1. Either the name of a function defined at the top level or a section occurs in an alternate of a multiple alternate expression in the body of the HOF, where some other alternate is the FVP to the HOF. Since the function name or section must be transformed into a constructor value (see Section 3.7.1), the constructor must be

```
        fun modify_fvp' fvp_expr fvp base_fvp scope env fvp_type =
change ▷    let val fvp_type = case fvp_type of
change ▷                           nothing ⇒ get_fvp_type fvp_expr env
change ▷                         | just t  ⇒ t
            in
                case fvp_expr of
                    Id nm ⇒
                        let val id_to_use = deref_id a env
                            val tag = store_fvp_info base_fvp id_to_use fvp_type
                        in
                           (Id (make_id ("P_" ^ id_to_use ^ "_") base_fvp tag, tag)
                        end
                  | Abs pelist ⇒
                        let val vars_in_section = section_vars pelist env
                            val (tag, section_num) = store_section_fvp_info base_fvp vars_in_section
                                                                            pelist fvp_type env
                            val rep = make_id ("P_Section" ^ section_num ^ "_") base_fvp tag
                        in
                            if vars_in_section = nil then
                                (Id rep, tag)
                            else
                                (App(Id rep, Tuple(map Id vars_in_section)), tag)
                        end
add ▷             | Case (cexp, (p_hd, e_hd) :: pelist) ⇒
add ▷                   let fun one_alt (p, e) =
add ▷                           modify_fvp' e fvp base_fvp scope (augment env p cexp) fvp_type
add ▷                       val (e'_hd, tag) = one_alt (p_hd, e_hd)
add ▷                       val pelist' = map (first ∘ one_alt) pelist
add ▷                   in (Case((p_hd, e'_hd) :: pelist'), tag) end
            end
```

Figure 5.5: *modify_fvp'*, which handles multiple alternate expressions

tagged with the specialization group tag. However, since the expression is within the body of the HOF, the tag must correspond to the tag used in each copy of the HOF. Thus, the FVP occurs in all specialization groups for the HOF set. For an example, consider a modified version of the previous example program.

$$
\begin{array}{ll}
\textbf{fun } \mathit{plus1}\ (x : \mathit{int}) & = x + 1 \\
\textbf{fun } \mathit{java}\ s & = s \hat{}\ \text{“}\mathit{java}\text{”} \\
\textbf{fun } \mathit{high}\ (f, x, \mathit{switch}) & = (\textbf{case } \mathit{switch} \textbf{ of } 0 \Rightarrow \mathit{id} \mid \_ \Rightarrow f)\ x \\
\textbf{val } \mathit{two} & = \mathit{high}\ (\mathit{plus1}, 1, 1) \\
\textbf{val } \mathit{coffee} & = \mathit{high}\ (\mathit{java}, \text{“}\mathit{mocca}\text{-”}, 1)
\end{array}
$$

The transformed version of this program follows. Note that a representation of *id* is included in both apply functions and FVP datatypes for *high*, and the representation is used in both copies of *high* with the appropriate tag in each case.

```
fun plus1 (x : int) = x + 1
fun java s = s ^ "java"
datatype type_1_of_high_A =
    P_plus1_1_of_high_A
    P_id_1_of_high_A
datatype type_1_of_high_B =
    P_java_1_of_high_B
    P_id_1_of_high_B
fun apply_1_of_high_A (function_rep, function_arg) =
    case function_rep of
        P_plus1_1_of_high_A ⇒ plus1 function_arg
      | P_id_1_of_high_A ⇒ id function_arg
fun apply_1_of_high_B (function_rep, function_arg) =
    case function_rep of
        P_java_1_of_high_B ⇒ java function_arg
      | P_id_1_of_high_B ⇒ id function_arg
fun high_A (f, x, switch) =
    apply_1_of_high_A (case switch of
                           0 ⇒ P_id_1_of_high_A
                         | _ ⇒ f, x)
fun high_B (f, x, switch) =
    apply_1_of_high_B (case switch of
                           0 ⇒ P_id_1_of_high_B
                         | _ ⇒ f, x)
val two = high_A (P_plus1_1_of_high_A, 1)
val coffee = high_B (P_java_1_of_high_B, "mocca-")
```

2. A recursive call to a HOF is made with a section FVP where the function called in the section is a parameter to the HOF. The constructor representing the section must be tagged with the specialization group tag. However, since the section is within the body of the HOF, the tag must correspond to the tag used in each copy of the HOF. Thus, the FVP occurs in all specialization groups for the HOF set. The transformation is similar to the one above.

5.3 Determining FVP Expression Types

The types of FVP expressions that are identifiers can be determined from the type information that is globally stored for each identifier. Type information is gathered at the beginning of the transformation process, by compiling the input program and using ML system functions to determine the types of all identifiers declared at the top level. This information is stored in a **ref** variable of type (*string* $*$ *string trep*) *list*, where the *string* is an identifier name, and the *string trep* is the type of that identifier. If *Firstify* were enhanced to allow for local function declarations in the input program, then type information would need to be gathered at each scope level.

The type of a FVP expression that is a section can be determined as follows. Consider the following section:

$$\lambda\, p.f(\ldots,\ \hat{p},\ \ldots)$$

The range of the section is the range of f. The domain of the section can be determined by examining the domain of f and determining the type of the argument in the position where $\hat{p}$ is placed. This type is the domain of the section.

If f is a function declared at the top level, then the type of f is stored in the type information list and so the type of the section can be determined. However, if f is a parameter to a surrounding function, then its type is not readily available. If the section occurs within the body of a HOF and f is a parameter to that HOF, then the section type does not need to be determined since the section will be represented in every FVP datatype for the HOF set, as described in Section 5.2. But if f is a parameter and the section is not in a HOF, then the type of f cannot be determined with the type information known to *Firstify*. This problem is exacerbated if f is a polymorphic function. The problem of determining the type of such sections remains unsolved.

Types of FVP expressions are calculated by the function *get_fvp_type*, shown in Figure 5.6. This function can calculate the type of functions declared at the top level and the type of sections, provided the function called in the section is a function declared

at the top level. It also can calculate the type of a multiple alternate expression, as described in the previous section.

```
fun get_fvp_type fvp_exp env =
    case fvp_exp of
        Id a ⇒ lookup_type(deref_id a env)
      | Abs [(p, App(f, arg))] ⇒
            let val ⟨⟨'d → 'r⟩⟩ = get_fvp_type f env
                val domain = get_type_at_pos (find_exp_pos (pat2exp p) arg) d
            in ⟨⟨'domain → 'r⟩⟩ end
      | Case(cexp, pelist) ⇒
            lgt (map (λ(p, e).get_fvp_type e (augment env p cexp))) pelist
      | Handle(hexp, pelist) ⇒
            lgt ((get_fvp_type hexp env) :: (map (λ(p, e).get_fvp_type e env) pelist))
      | ExpTypeC(e, t) ⇒ t
```

Context	
Identifier	**Type**
	Description
find_exp_pos	*string erep → string erep → parameter_id* Returns the tuple position of the first expression in the section expression.
get_type_at_pos	*parameter_id → string trep → string trep* Returns the type at the given tuple position within another type.
lgt	*string trep list → string trep* Returns the least general type of a list of types.
lookup_type	*string → string trep* Returns the type of an identifier, according to the information stored in the type information list.
pat2exp	*string pattern → string erep* Converts a pattern to an expression.

Figure 5.6: *get_fvp_type*

Chapter 6

Other Problems

This chapter discusses some unsolved or partially solved problems in the *Firstify* implementation.

6.1 Functions Contained in Data Structures

As has been noted, *Firstify* does not transform function values contained in data structures other than tuples. In particular, *Firstify* cannot handle functions contained in ML records or datatype values.

Extending *Firstify* to transform functions contained in records would be a straightforward process, since records are very similar to tuples. The mechanism for maintaining environment information would need to be extended to keep track of field tags, so that when a function value is "extracted" from the record, the field in which it was contained is stored in the environment. The *parameter_id* type would also need to be extended to track the positions of function values in records.

It is believed that similar extensions can be made to *Firstify* to handle functions contained in datatype values. However, an alternative method of transforming such functions would be to treat a constructor with a functional domain like a HOF, and transform it as such. The transformations would be very similar to those currently done on "real" HOFs and calls to them. For example, consider the following program fragment.

```
datatype Changes =
    ChangeWith of int → int
  | Unchanged

fun change_number (change, x) =
    case change of
        ChangeWith f ⇒ f x
      | Unchanged ⇒ x

val two = change_number (ChangeWith increment, 1)
```

In this program fragment, the argument to the constructor *ChangeWith* is a function. If *ChangeWith* is treated as a HOF (and hence (*ChangeWith*, *Position 0*) is a FVP), then the following transformations are needed:

1. An FVP datatype and apply function must be constructed to represent functional arguments to *ChangeWith*.

2. Each time a functional argument to *ChangeWith* is applied, the application must be transformed into a call to the apply function.

3. Each "call" to *ChangeWith* (*i.e.* applications of *ChangeWith* to an argument) must be transformed so that the argument is a value in the FVP datatype.

This results in the following transformed program fragment.

```
datatype type_0_of_ChangeWith =
    P_increment_0_of_ChangeWith

datatype Changes =
    ChangeWith of type_0_of_ChangeWith
  | Unchanged

fun apply_0_of_ChangeWith (function_rep, function_arg) =
    case function_rep of
        P_increment_0_of_ChangeWith => increment function_arg

fun change_number (change, x) =
    case change of
        ChangeWith f => apply_0_of_ChangeWith (f, x)
      | Unchanged => x

val two = change_number (ChangeWith P_increment_0_of_ChangeWith, 1)
```

Some complications to this approach include:

- The domain of the constructor needs to be explicitly modified (as in *Changes*, above).
- Constructors can have shared FVPs with other constructors or with HOFs.
- To specialize constructor "HOFs," the datatype needs to be specialized. In the above example, no specialization is necessary since the function type is mandated to be of type $int \rightarrow int$, but this is not true in general.

There is another unsolved problem relating to functions contained in data structures. *Firstify* does not transform functions that are "extracted" from a data structure via a function call. This is true for tuples as well as other data structures. For example, consider the following declaration fragment of a HOF *high*.

```
fun high (twofuns as (f, g), x) =
    ... (first twofuns) x ... (second twofuns) x ...
```

If we know that *first* and *second* are the first and second projection from a pair, then it is clear that both f and g are FVPs. However, *Firstify* has no means to determine this. It appears that the only solution to this problem involves analysis of the declarations of "extraction" functions like *first* and *second*.

6.2 HOFs with Higher Order FVPs

Consider the following program fragment:

```
fun high (f, x) = f x
fun higher (f, g, x) = f (g, x)
val call_higher = higher(high, id, 1)
```

Both *high* and *higher* are HOFs. However, *high* is never explicitly called. Although *high* will be detected as a HOF, no functions are explicitly passed as FVPs to *high*. Analysis reveals that *id* is a FVP for *high*, and so *id* should be represented in the FVP datatype for *high* and interpreted in the apply function for *high*. But without an explicit call to *high*, the FVP datatype and apply function for *high* are not created. If we run *Firstify* on the program fragment, we would get something like the following:

```
datatype type_1_of_high =
    ?
fun apply_1_of_high (function_rep, function_arg) =
    case function_rep of
        ?
datatype type_1_of_higher =
    P_high_1_of_higher
fun apply_1_of_higher (function_rep, function_arg) =
    case function_rep of
        P_high_1_of_higher ⇒ high function_arg
fun high (f, x) = apply_1_of_high (f, x)
fun higher (f, g, x) = apply_1_of_higher (f, (g, x))
val call_higher = higher(P_high_1_of_higher, id, 1)
```

Notice that there is now an explicit call to *high* in *apply_1_of_higher*. Thus, a solution to this problem would be make sure that we execute Phase III and Phase IV for (*high*, *Position 1*) after executing Phase IV for (*higher*, *Position 1*). Thus, the call to *high* would be detected, a representation for *id* would be included in *type_1_of_high*, and the identifier *id* in the declaration of *call_higher* would be transformed into the identifier *P_id_1_of_high*[1]. In this example, if (*higher*, *Position 1*) is detected before (*high*, *Position 1*), all this will take place. But since we cannot guarantee the order in which HOFs are detected, we need a more general solution.

The proposed solution is to look for untransformed HOF calls after *all* HOFs have been transformed. This would be performed in two new phases, Phase V and VI, which would be similar to Phases III and IV. A diagram illustrating this modified phase execution order is shown in Figure 6.1.

6.2.1 Phase V—Transforming HOF Calls in Apply Functions

This phase is similar to Phase III, except that only apply functions are examined for calls to HOFs. When a HOF call is found in any apply function, the apply function will be detected as HOF, with a FVP shared with the HOF being called. This is desirable because the functions passed as FVPs will be passed to the apply function, so calls to the apply function will need to be transformed.

This phase would require that functions contained in datatype values could be detected and transformed, as discussed in Section 6.1. This is because HOF calls in apply functions might use as the FVP a function contained in a datatype value, where the datatype value is the representation of a section.

---

[1]This last transformation would occur because (*high*, *Position 1*) is shared with (*apply_1_of_higher*, *NestedAt* (*2*, *Position 1*)) which is shared with (*higher*, *Position 1*).

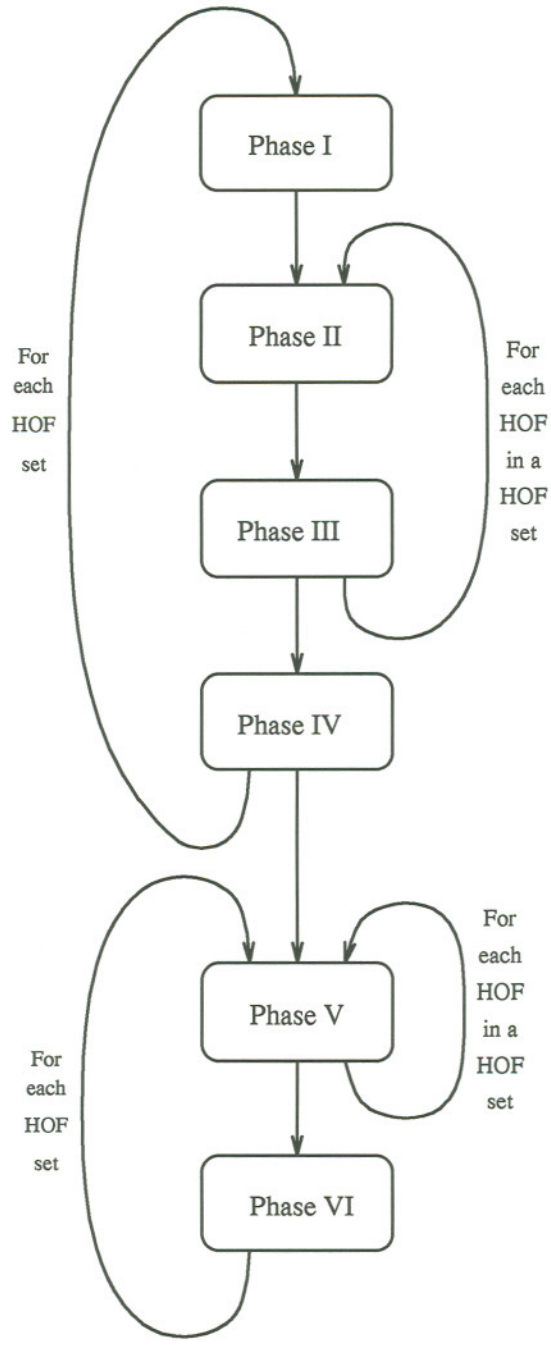

Figure 6.1: Phase execution order for dealing with HOF calls in apply functions

6.2.2 Phase VI—Respecialization and Adding/Modifying Apply Functions

If a new HOF call is detected in Phase V, a constructor representing the FVP function needs to be added to an apply function and FVP datatype for the HOF. If no other calls to the HOF exist (as in the example above) then this is straightforward. However, if other HOF calls exist, the constructor must be added to the appropriate FVP datatype and corresponding apply function based on the specialization group of the FVP (*i.e.* the type of the FVP expression).

When a HOF call is found, it may also be discovered that the FVP is shared with other HOFs that were previously transformed. This requires that the FVP datatypes and apply functions for the two HOFs be merged. For example, consider the following modified version of our original example:

$$\begin{aligned}
&\textbf{fun } \mathit{high}\ (f, x) \ = \ f\ x \\
&\textbf{fun } \mathit{higher}\ (f, g, x) \ = \ (f\ (g, x), g\ x) \\
&\textbf{val } \mathit{call_higher} \ = \ \mathit{higher}(\mathit{high}, \mathit{id}, 1) \\
&\textbf{val } \mathit{two} \ = \ \mathit{high}(\mathit{increment}, 1)
\end{aligned}$$

In this example, (*high*, *Position 1*), (*higher*, *Position 1*), and (*higher*, *Position 2*) will all be detected as HOFs in Phase I. However, it will not be detected that (*high*, *Position 1*) is a shared FVP with (*higher*, *Position 2*) until phase V, after FVP datatypes for both these FVPs have been created. Thus, the two datatypes and apply functions corresponding to them must be merged. Merging FVP datatypes and apply functions must take into account specialization groups so that the resulting program is well-typed.

6.2.3 The Current Implementation

Currently, *Firstify* does not implement the solution described in this section. Instead, it analyzes the input program before executing phase I and "seeds" the transformation list with HOFs that have HOFs as parameters. These are detected by examining the type of each function. By seeding the transformation list, it is ensured that these functions will

be the first ones transformed, and so calls to HOFs in apply functions will be introduced as early as possible.

During this analysis phase, the bodies of HOFs with higher order FVPs are also examined, looking for applications of the higher order FVP with some other parameter as a FVP to the higher order FVP. If such applications are found, they are recorded. This information is used as follows. Suppose a higher order function h has a higher order parameter g, passed to h in position P_1. Analysis of h shows that another parameter to h (say $(h, Position\ P_2)$) is a FVP to g, passed to g in position P_3. As calls to h are transformed, the function passed as the higher order FVP (in position P_1) is recorded in the transformation list as a HOF. Furthermore, this HOF shares its FVP in position P_3 with $(h, Position\ P_2)$, and this information is also recorded in the transformation list.

This technique will correctly transform many HOFs that have HOFs as FVPs. However, the analysis is incomplete, and will not work in two situations:

1. If the function is polymorphic in the domain of the FVP that is a HOF, then the analysis does not work, because it cannot tell that the FVP is higher order.

2. If the function has a FVP that is a higher order function that has a FVP that is a higher order function, the analysis does not work. This problem extends to further levels–that is, higher order functions with higher order FVPs that are higher order functions with FVPs that are higher order functions, and so on.

It is believed that the analysis can be extended to detect functions in the second situation. However, even with this extension the analysis will still fail if the function is polymorphic in the domain of the FVP that is a HOF. The only solution to this appears to be requiring that higher order functions are monomorphic (which would obviate the need for specialization as well).

6.3 HOF Call Site Limitations

The current implementation of *Firstify* assumes that at the HOF call site the HOF expression is an identifier or multiple alternate expression with each alternate an identifier

(or another multiple alternate expression). Also, the FVP expression must either be an identifier or a section, or a multiple alternate expression with all alternates being identifiers or sections or another multiple alternate expression. This enables *Firstify* to make the transformations to the HOF name (tagging with a specialization group tag) and FVP expression (transforming into a data value) at the call site.

This seems to be a reasonable restriction, particularly because of the assumption that there are no **let** declarations in the input program. If **lets** were allowed, then transformations to HOF names or FVP expressions might be necessary in a **let** declaration preceding a HOF call site rather than at the call site. This is because the FVP or HOF expression could be an identifier declared locally that refers to another expression, and the other expression would require transformation.

However, even without **lets**, the problem remains. Because of **val** declarations at scope level 0 and certain uses of **case** expressions, it is possible for an identifier at the call site to refer to another expression that would require transformation. The following five instances of this problem have been identified.

1. The parameter to the HOF is an identifier declared at scope level 0, and the identifier refers to a tuple containing the FVP expression and other values. Example:

$$\begin{aligned} &\textbf{val}\ params\ =\ (increment, 1) \\ &\quad\vdots \\ &\textbf{val}\ call\_high\ =\ high\ params \end{aligned}$$

2. The HOF call is in an alternate of a **case** expression, and the parameter to the HOF is the pattern for that alternate. The selection expression is a tuple containing the FVP expression and other values. Example:

$$\textbf{case}\ (increment, 1)\ \textbf{of}\ params\ \Rightarrow\ high\ params$$

3. The HOF call is in an alternate of a **case** expression. The selection expression is another multiple alternate expression, each alternate of which is a HOF expression.

The pattern for the alternate containing the HOF call is used as the HOF expression. Example:

$$\textbf{case}\ (\textbf{case}\ e\ \textbf{of}\ P_1\ \Rightarrow\ high_1\ |\ P_2\ \Rightarrow\ high_2)\ \textbf{of}$$
$$high\ \Rightarrow\ high\ (increment, 1)$$

4. The HOF call is in an alternate of a **case** expression. The selection expression is another multiple alternate expression, each alternate of which is a FVP expression. The pattern for the alternate containing the HOF call is used as the FVP expression. Example:

$$\textbf{case}\ (\textbf{case}\ e\ \textbf{of}\ P_1\ \Rightarrow\ increment\ |\ P_2\ \Rightarrow\ decrement)\ \textbf{of}$$
$$fvp\ \Rightarrow\ high\ (fvp, 1)$$

5. The HOF call is in an alternate of a **case** expression. The selection expression is another multiple alternate expression, each alternate of which is a tuple containing the FVP expression and other values. The pattern for the alternate containing the HOF call is used as the parameter to the HOF. Example:

$$\textbf{case}\ (\textbf{case}\ e\ \textbf{of}\ P_1\ \Rightarrow\ (increment, 1)\ |\ P_2\ \Rightarrow\ (decrement, 3))\ \textbf{of}$$
$$params\ \Rightarrow\ high\ params$$

It is unclear what the best solution to instance 1 might be. It appears that any solution would require major changes to the *Firstify* implementation to accommodate the transformation.

A partial solution to the other instances of the problem is known. Notice that the outermost **case** expression in each example contains only one alternate. As a result, the **case** expression can be transformed by replacing the pattern occurring in the alternate expression with the selection expression. Examples in instances 2–5 would be transformed into:

2. $high\ (increment, 1)$

3. $(\textbf{case}\ e\ \textbf{of}\ P_1\ \Rightarrow\ high_1 \mid P_2\ \Rightarrow\ high_2)\ (increment, 1)$

4. $high\ ((\textbf{case}\ e\ \textbf{of}\ P_1\ \Rightarrow\ increment \mid P_2\ \Rightarrow\ decrement), 1)$

5. $high\ (\textbf{case}\ e\ \textbf{of}\ P_1\ \Rightarrow\ (increment, 1) \mid P_2\ \Rightarrow\ (decrement, 3))$

Each of these HOF calls can be transformed with the current *Firstify* implementation.

This solution is incomplete. If *Firstify* were extended to allow functions contained in datatype values, then this solution would not work because the outermost **case** expression could then have more than one alternate.

Chapter 7

Summary

Firstify is an implementation of Reynold's method for eliminating higher order functions from a functional language program. *Firstify* transforms a restricted set of Standard ML programs expressed in the abstract syntax of CRML into semantically equivalent programs that are first order.

The work described in this thesis addresses features of modern functional programming languages not explicitly addressed in Reynold's paper or other references. These features include pattern matching, parametric polymorphism, multiple alternate statements, local declarations, and function values contained in data structures. This work also elaborates details left out of earlier presentations of the method, including detection of higher order functions, function values that are "shared" by several higher order functions, and higher order functions with higher order parameters.

7.1 Input Assumptions

Not all of the above features and details are implemented in *Firstify*. However, assuming other relatively minor transformations to a program are performed before using *Firstify*, programs using a significant portion of the core ML syntax are correctly transformed by *Firstify*. Assumptions about programs transformed by *Firstify* include the following:

1. The program contains no **let** expressions. This can be ensured with a transformation that "lifts" local declarations to the top level.

2. All functions in the program are "strongly uncurried"—that is, each function has a single argument (which may be a tuple of values). This can be ensured with another transformation.

3. The only lambda abstractions that exist are sections that are passed as function values to higher order functions. This can also ensured via the uncurrying transformation.

4. All identifiers declared at the top level are unique.

5. No identifiers declared in the program conflict with identifiers introduced by *Firstify*. See Section 3.5 for more information.

6. At call sites of higher order function, the HOF expression must be either an identifier referring to a function or a multiple alternate expression with each alternate an identifier (or another multiple alternate expression). Also, the FVP expression must either be an identifier or a section, or a multiple alternate expression with all alternates being identifiers or sections (or another multiple alternate expression).

7. Functions are not "contained" in record values or datatype values.

8. Higher order functions with higher order parameters are restricted, as discussed in Section 6.2.

9. All sections have a single pattern and expression.

10. Issues concerning **raise** and **handle** have not been explored, except for treating **handle** expressions as multiple alternate. Thus, it is assumed that the input program does not use exceptions.

7.2 Future Work

Future work includes eliminating or weakening the above assumptions. It is particularly desirable to extend *Firstify* as follows:

1. Weaken the assumption about uncurrying to the assumption that all function declarations and function calls include all arguments to the function, but the arguments are not necessarily contained in a tuple. Making this assumption would require that *Firstify* perform "strong" uncurrying on functions passed as FVPs, but not on other functions.

2. Ensure that identifiers introduced by *Firstify* do not conflict with existing identifiers.

3. Extend *Firstify* to handle functions "contained" in datatype values.

4. Allow sections with multiple alternates.

5. Extend *Firstify* to handle the module system of Standard ML with a version of CRML that supports the module system. Presumably *Firstify* would then operate on signatures and structures rather than lists of declarations.

Another topic for future work concerns specialization and monomorphism. Currently, specialization is used to make higher order functions and their corresponding apply functions monomorphic with respect to their functional parameters. In Section 6.2 it is shown that the analysis to detect higher order functions with higher order parameters (which occurs before specialization) is stymied when such functions are polymorphic. Therefore, a more effective implementation might "monomorphize" input programs before applying Reynold's method.

Appendix A

The *enventry* Datatype Constructors

Constructor	**Domain**
	Description of Use
	Example
NotInEnv	()
	Default *enventry* value, for unbound identifiers
	None
NotApplicable	()
	Identifier without a meaningful binding to the transformation process, such as a constant
	let val x $=$ 1 **in** *BODY* **end** While processing *BODY*, x is bound to *NotApplicable*.
RefTo	*string*
	Identifier directly referring to another identifier
	let val x $=$ y **in** *BODY* **end** While processing *BODY*, x is bound to *RefTo "y"*.

Constructor	**Domain**
	Description of Use
	Example
Param	*function_id* $*$ *parameter_id*
	Part of a parameter to a function. The *function_id* encodes which function, and the *parameter_id* encodes the position of the identifier in the function's parameter.
	fun $f\ x\ =\ BODY$ While processing *BODY*, x is bound to *Param* ((*"f"*, 0), *Position* 0). The types *function_id* and *parameter_id* are discussed in Section 3.3.1.
LocalDec	*function_id*
	Identifier declared in a **val**, **fun**, **datatype**, **exception**, or **type** declaration. The *function_id* encodes information about the identifier, including the scope at which it is declared.
	fun $f\ \ldots$ While processing all declarations occurring after the declaration of f, f is bound to *LocalDec* (*"f"*, 0). The *function_id* type is discussed in Section 3.3.1.
TuplePartOf	*int* $*$ *enventry*
	Identifier that is contained in a tuple. The *enventry* represents the tuple in which the identifier is contained, and the *int* encodes the position of the identifier in the tuple.
	val $z\ =\ a$ **let val** $(x, y)\ =\ z$ **in** *BODY* **end** While processing *BODY*, x is bound to *TuplePartOf* (1, *RefTo "a"*) and y is bound to *TuplePartOf* (2, *RefTo "a"*).

Constructor	**Domain**
	Description of Use
	Example
ConstrPartOf	*enventry*
	Identifier to which a constructor is applied. The *enventry* represents the value in the datatype, that is, the value of a constructor applied to the identifier.
	val z = a **let val** $Cons(x,y)$ = z **in** *BODY* **end** While processing *BODY*, x and y are each bound to *ConstrPartOf* (*RefTo* "a").
RecordPartOf	*enventry*
	Identifier that is within a record. The *enventry* represents the value in the record type, where the identifier is contained in some field of the record value.
	val z = a **let val** $\{t_1 = x, t_2 = y\}$ = z **in** *BODY* **end** While processing *BODY*, x and y are each bound to *RecordPartOf* (*RefTo* "a").
BasedOn	*enventry*
	Identifier that is the result of a function application to another value. The *enventry* represents the value to which the function is applied. This constructor is not used in the current implementation of *Firstify*.
	val z = a **let val** x = $f\ z$ **in** *BODY* **end** While processing *BODY*, x is bound to *BasedOn* (*RefTo* "a").

Constructor	**Domain**
	Description of Use
	Example
TupleContaining	*enventry list*
	Identifier is a tuple, containing several values. The values within the tuple are represented in order in the *enventry list.*
	val $y = a$ **val** $z = b$ **let val** $x = (y, z)$ **in** *BODY* **end** While processing *BODY*, x is bound to *TupleContaining* [*RefTo "a", RefTo "b"*].
RecordContaining	(*string* $*$ *enventry*) *list*
	Identifier is a record, containing several values. Each (*string*, *enventry*) pair represents a value within the record. The *string* is a field tag and the *enventry* represents the actual value.
	val $y = a$ **val** $z = b$ **let val** $x = \{t_1 = y, t_2 = z\}$ **in** *BODY* **end** While processing *BODY*, x is bound to *RecordContaining* [(*"t_1", RefTo "a"*), (*"t_2", RefTo "b"*)].

Constructor	**Domain**
	Description of Use
	Example

PossibleValues	*enventry list*
	Identifier refers to several possible values, such as when a **case** or **handle** expression is used. Each *enventry* in the list represents one possible binding for the identifier.
	val y = a **val** z = b **let val** x = **case** c *of* $0 \Rightarrow y$ $\mid \_ \Rightarrow z$ **in** *BODY* **end** While processing *BODY*, x is bound to *PossibleValues* [*RefTo* "a", *RefTo* "b"].

Constructor	**Domain**
	Description of Use
	Example
Closure	$environ * int * ((pattern * (environ \rightarrow enventry))\ list)$
	Identifier refers to a lambda abstraction. The *environ* is the environment in which the abstraction is defined. The *int* is the scope level (see Chapter 4) at which the abstraction is defined. The list of *patterns* and *environ* $\rightarrow$ *enventry* functions provides a means for evaluating the closure when the lambda abstraction is applied. The environment value of an application $f\ x$, where f has the environment value *Closure*(*def_env*, *def_scope*, *pelist*) and x has the environment value x_{env} can be calculated as follows: $PossibleValues\ (map^{list}\ (\lambda(p,e).\ e\ (def\_env \mid p \rightarrow x_{env}))\ pelist)$ Because of assumed restrictions to the input of *Firstify*, this constructor is not used in the current implementation.
	let val $f\ =\ \lambda\ x.x+1$ **in** $BODY$ **end** While processing $BODY$, x is bound to $Closure\ (env, s, [(\langle\langle x\rangle\rangle, bf)])$, where env is the current environment, s is the current scope, and bf is a function which if passed an environment will return an environment value for the body of the lambda abstraction.

Appendix B

Values Referenced in Accumulator Functions

Non-function Values	
Identifier	**Type**
	Description
base_fvp	*fparm*
	The base FVP for the current HOF set.
fvp	*fparm*
	The current FVP, for which transformations are being performed.
scope	*int*
	Current scope level.

Function Values	
Identifier	**Type**
	Description
add_xform_fun	*fparm* → *fparm list* → *int* → () Adds a transformation to the transformation list. The *fparm* is the base HOF, and the *fparm list* is the HOFs with shared FVPs (which might be just the base FVP). The *int* is the scope at which the base HOF is defined.
augment	*environ* → *string pattern* → *string erep* → *environ* Augments an environment with bindings created by binding the pattern to the environment value of an expression.
deref_id	*string* → *environ* → *string* "Dereferences" an identifier name by looking it up in the environment. If the identifier refers to some other identifier, returns the other identifier's name. If not, returns the original name.
enrich	*environ* → *string pattern* → *enventry* → *environ* Augments an environment with bindings created by binding the pattern to an environment value.
extract_param_env	*enventry* → *parameter_id* → *string erep* Extracts an environment value from a environment value for a tuple expression at the given position.
extract_param_exp	*string erep* → *parameter_id* → *string erep* Extracts an expression from a tuple expression at the given position.

Function Values	
Identifier	**Type**
	Description
get_function_info	*enventry* → *environ* → *function_id list* Returns a list of functions referenced by an environment value.
get_param_info	*enventry* → *function_id list* Gets information about parameters referenced in an *enventry* value.
insert_param_exp	*string erep* → *string erep* → *parameter_id* → *string erep* Inserts an expression into a tuple expression at the given position.
lookup_enventry	*environ* → *enventry* → *enventry* Looks up an environment value by chasing pointers to other environment values.
make_id	*string* → *fparm* → *string* → *string* Constructs an identifier based on a prefix string, a FVP, and a tag string.
refs_param	*enventry* → *bool* Indicates whether an environment value references a parameter.

Bibliography

[BJ82] Dines Bjørner and Cliff B. Jones. *Formal Specification and Software Development*, pages 277–280. Prentice-Hall, 1982.

[CD93] Wei-Ngan Chin and John Darlington. Higher-order removal: A modular approach. Unpublished work, 1993.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Patterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, Cambridge, MA, August 1991. Springer Verlag. Lecture Notes in Computer Science Volume 523.

[Nel91] George C. Nelan. *Firstification.* PhD thesis, Arizona State University, December 1991.

[Pau91] Laurence C. Paulson. *ML for the Working Programmer*, pages 377–381. Cambridge University Press, 1991.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740. ACM, 1972.

[Sch85a] David A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.

[Sch85b] David A. Schmidt. An implementation from a direct semantics definition. *Lecture Notes in Computer Science*, 217:222–235, 1985.

[Sch88] David A. Schmidt. Detecting stack-based environments in denotational definitions. *Science of Computer Programming*, 11(2):107–131, December 1988.

[She93] Tim Sheard. Guide to using CRML, Compile-Time Reflective ML. Technical report, Oregon Graduate Institute of Science and Technology, 1993.

[Wan85] Mitchell Wand. From interpreter to compiler: A representational derivation. *Lecture Notes in Computer Science*, 217:306–324, 1985.